TEX and Scripting Languages

William M. Richter
Texas Life Insurance Company
900 Washington Avenue, Waco, TX 76703, USA
wrichter@texaslife.com

Abstract

TEX is an ASCII text-based markup language. In a scheme of automated document preparation TEX provides the foundation. The idea is for programs to do the work of 1) generating the TEX code for documents, 2) running TEX on these documents, and 3) post-processing the resulting `.dvi` files to obtain the finished documents. Resulting PostScript documents may be further post-processed to produce files that exploit the output capabilities of various printers. Discussed herein are the techniques and benefits of such a scheme and how scripting languages (those languages outside the traditional edit/compile/link/run cycle) can make the whole process fun and easy.

1 Introduction

In his web essay *Hackers and Painters* [2], Paul Graham equated the much maligned and misunderstood activity of "hacking" [6] with the long-esteemed tradition of painting (e.g., portrait painting, as opposed to painting of porches, peeling house trim, and such). He observed that what, today, we acknowledge as masterworks actually evolved during the artist's act of creation from a sketch, the details only gradually being filled in, to a finished, glorious work of art. He argued that a writer goes through the same process of refinement, starting from rough outline or foggy idea until she finds nothing which needs refining. One reason TEX is appealing to authors is that it makes the process of refinement secondary. The tasks of *creation* (thinking is hard work for most of us) and *presentation* are orthogonal. Moreover the presentation task is assumed almost entirely by TEX.[1] One can, after all, create a TEX document that is 90% complete using nothing more than a tool as simple as NotePad. The implication being that simple tools equate to less loss of creative energy.

Graham believes that authors of computer code (programmers, we often call them) follow the same nonlinear/circuitous paths of painters and authors. Seldom, if ever, is software conceived of and implemented by following in a direct route from beginning to end. Most great software, Graham claims, is the product of hacking, that the implications for software design are significant, and that what a computer language is and how an author interacts with it defines the end result. In his view it means . . .

> . . . a programming language should, above all, be malleable. A programming language is for thinking of programs, not for expressing programs you've already thought of. It should be a pencil, not a pen.

And he continues,

> We need a language that lets us scribble and smudge and smear, not a language where you have to sit with a teacup of types[2] balanced on your knee and make polite conversation with a strict old aunt of a compiler.

A class of programming languages, called "scripting languages", is compatible with Graham's ideas of what a hacker's language should be. "Malleable" in nature, and easy to *think with*, scripting languages are similar in spirit to TEX. Indeed, TEX itself may even be considered as a scripting language for typesetting.

So, on the one hand, we have TEX, a tool which lets authors "scribble and smudge and smear" about with their ideas. On the other hand we have hackers using scripting languages pursuing similar creative avenues. The question then arises, "What happens if these two tools are combined and used in a collaborative effort?" We now explore various ways that TEX and scripting languages can be combined.

[1] Except when we TEXnicians decide we know better and begin to muck around in TEX's own internal affairs.

[2] For readers unfamiliar with the art of computer programming, the "teacup of types" to which he is referring will be addressed in a subsequent section on the attributes of scripting languages, where static vs. dynamic data types are discussed.

2 Scripting Languages

Before delving into scripting languages proper, let us review a few of the attributes of traditional computer languages (the ones compiled with Paul Graham's strict old aunts).

3 Traditional Computer Languages

For readers unfamiliar with the art of authoring computer software (programming computers), here is what programmers do: they think of a task that computers can accomplish better than humans (say, typesetting, for example). Then they sit and think, potentially at length, about how humans would go about doing that task, and how to express those steps algorithmically [3]. After sketching said algorithm, they formalize and codify it in a so-called "language" that is a sort of half-way meeting ground between the way humans think and the way computers operate. This prose, called a program, consists of two distinct entities: variables, which declare what it **is** that the computer will be working on, and imperative procedures that define what is to be **done** to that data.

Some salient details about these traditional languages:

1. The variables: Computer hardware can work with data in different formats: numbers (integers and real numbers), strings of character data, etc. Each variable in a program must be defined in advance of its use to be of a specific type. In computer science lingo this is called *static typing.*
2. The code: Codifying an algorithm in a particular computer language isn't really enough for computer hardware. More work must be done. This language must be *compiled* by Graham's "aunt" into "machine code" on which the computer's logic circuits can act.
3. But even the work of the compiler-aunt isn't enough. The fruit of her strict dominance must then be *linked* with the work of other compiler-aunts to produce a final collection of unreadable "goo" that only a computer can understand (machine code is unreadable to all but the most deviant of human brains).
4. Nor is this the end of the story. When an edited/compiled/linked program (called an executable) has finally been produced and a blazingly fast 3-gigahertz CPU is unleashed to execute it the first time, the most likely end result is either an almost immediate decision by the CPU that its human programmer is capable only of producing flawed code for it to execute (it communicates this fact by printing some rude message like **"Segmentation Violation"** and producing a huge file on disk containing the entire contents of its memory), or it lapses into a seemingly semi-comatose state consuming large amounts of CPU time until its programmer/master gets its attention with the violence of the `kill` command.

One can see a definite "cycle of pain": *Edit, Compile, Link, Test* that must be repeated many times until a flawless executable is produced. No wonder computer programming is seen by many an outsider as a black art to be pursued by only the most intrepid and determined souls.

4 Why Scripting Languages are Better, and Why More People Should be Hackers

Scripting languages [9] shrink the cycle of pain to *Edit, Test.* With the crufty old compiler-aunt gone, the whole process of software development proceeds in a more efficient and pleasant manner with attention shifting to the "creative", editing part and the refinement, or testing part. But measure of pain is not the only attribute that makes scripting languages attractive. Other important attributes are:

1. Simple syntax;
2. High-level data types;
3. Loosely typed;
4. Standard control structures: if/else, while, for;
5. Interfaces well with host operating system;
6. Plays well with external entities;
7. Embeddable inside more complex systems;
8. Often used as "glue" languages to link multiple standalone applications and tools together;
9. Requires a runtime interpreter to execute the script;
10. Compiles to bytecode which executes on a virtual machine;
11. Often 'dynamic' in nature.

We need to expound on a few of these points.

4.1 Simple Syntax

If a language is to satisfy Graham's requirement that it be a malleable pallet for the smearing and smudging of ideas, it cannot be verbose (we don't want to spend time typing). So scripting languages (henceforth SLs — I'm tired of typing, too) are succinct in nature; able to convey a significant amount of procedural instruction in as few words as necessary to maintain clarity of meaning.[3]

[3] For programming language scholars, the language APL may come to mind, but perhaps not *that* succinct. It would

4.2 High-level data types

The concept of high-level data types parallels simple syntax. Just as we need to state procedural algorithms in a succinct fashion, we also need constructs that allow for the representation of bundles of data that may be arbitrarily complex. We demand more than simple integer, floating point, and strings of character data that traditional languages like C and C++ provide.[4]

Usually these higher-level data types come in the form of lists and dictionaries; containers that hold other data elements and allow for the expression of relationships between our data.

4.3 Loosely Typed / Dynamic Nature

Discussion of esoteric topics like *Strongly vs. Loosely Typed Data* and *Early vs. Late Binding* is more than can be discussed here (see [1]). Some understanding is essential, however. Earlier, we pointed out that in traditional languages, each element of data that a program will use (its variables) must be defined to exist as a particular type before it can be used.[5] Moreover, as variables are passed between parts of a program (function calls) the type of each variable passed must match exactly the type expected by the called function. This check is done by the strict old compiler-aunts, and was designed to keep programmers from making errors that would only manifest themselves during the test phase. Strict type checking makes a lot of sense with traditional languages.

However, with dynamic SLs, there is a critical difference, rooted in the 'dynamicness' of the language. SLs need not declare variables in the first place. Variables are created or 'allocated' (on-the-fly, as it were) when they are first referenced. When a variable is allocated it is associated with a particular type that is implied from the context in which it was initially used. The association to type is permanent and observable. So not only can one ask, "What *value* does a variable contain?", one can also make an inquiry about its *type*.

For example, the statement `A = 123` allocates a data element called `A` whose value is 123 and whose type is integer. The statement `B = 3.14` allocates a variable called `B` whose value is 3.14 and whose type is floating point. `B` was made a variable of type floating point because, contextually, the statement contained a decimal point in the value implying a floating point value. Had we desired `A` to be a floating point variable we would have coded `A = 123.0`.

This leads to a new world of ways in which to think about writing code. Functions, now dynamic in nature, can easily accept an arbitrary number of arguments, the type of each being any of a range of possible types. Depending on the number and type of variables passed to a function, the function may act in different ways. This goes to the heart of malleability. In the creative process if we change our mind and decide to "smudge and smear" in a different direction, our existing code may not go to waste. It may be possible just to extend it to conform to our new conditions.

A world of new and easier programming languages, the SLs, may also introduce hacking to a wider audience. Whereas the "old world" traditional languages excluded or intimidated many people for the reasons above (there are, after all, only so many work hours in a day), SLs remove the complexity of programming and make hacking the creative process that it should be.

Finally, there is another reason more people (at least for those who must live with a computer) should become hackers. While most of us are not master software developers, developing cathedral-size financial accounting packages, for example, we do a surprising amount of "sketch" work (in Graham's paradigm) and having skills to write small programs can be effective.

5 Real Scripting Languages

A mid-June 2004 *google-search* for the keywords
`script language programming`
returned approximately 1,570,000 hits. Top-ranked pages returned from a search of keywords `scripting languages` reside on the sites:

1. `www.php.net`
2. `www.python.org`
3. `www.ruby-lang.org`
4. `www.perl.org`

All these websites are homes of important scripting languages. And there are more SLs; many more ... a veritable zoo, with names like: Awk, JavaScript, Lisp, Lua, Perl, PHP, Python, Rebol, Ruby, Small, Groovy, Tcl. If one were to rank SLs in order of popularity, the top of that list would include:[6]

- Perl
- Python

be nice for non-hackers to be able to read and understand our prose, too.

[4] Admittedly, C, C++, and other traditional languages may be made to represent arbitrarily complex data, but those types are not intrinsic in the language.

[5] This isn't actually true. Data elements may be dynamically allocated in traditional languages, but this introduces additional complexity in both the design and debugging steps.

[6] Not listed in order.

- Tcl/Tk
- JavaScript
- Unix shell scripts (`sh`/`bash`/`csh`/etc.)

Several of these SLs have outgrown their scripting origins and have gone on to become "general purpose programming languages of considerable power" [5]. The only argument for continuing to use the term "scripting language" is the lack of a better term.

6 A Particular Scripting Language: Python

Chapter one of the official Python Tutorial reads:[7]

> Python is simple to use, but it is a real programming language, offering much more structure and support for large programs than the shell has. On the other hand, it also offers much more error checking than C, and, being a very-high-level language, it has high-level data types built in, such as flexible arrays and dictionaries that would cost you days to implement efficiently in C. Because of its more general data types Python is applicable to a much larger problem domain than Awk or even Perl, yet many things are at least as easy in Python as in those languages.

The tutorial continues to highlight these important attributes:

1. It has a modular architecture so that code developed for one application can be reused in other programs. Likewise, it comes with a large number of built-in modules for things like file I/O, system calls, sockets, and many common Internet protocols (FTP, HTTP, SMTP, etc.).
2. It is an interpreted language conforming to the edit/test cycle discussed previously.
3. Its interpreter can be used interactively, making it easy to experiment with features of the language, or to test code before actually running a program (we'll see an example later, in fig. 11).
4. It has a high-level syntax that allows for writing compact, readable programs.
5. It has high-level data types allowing for expressions of complex data relationships.
6. It is object-oriented [10], but does not require the use of those object-oriented features, or O-O programming skills to use the language.
7. Statement grouping is done by indentation instead of explicit begin/end brackets.
8. It is extensible: if you know how to program in C it is easy to add a new built-in function or module to the interpreter, either to perform critical operations at maximum speed, or to link Python programs to libraries that may only be available in binary form (such as a vendor-specific graphics library).
9. It is embeddable: You can link the Python interpreter into an application written in C and use it as an extension or command language for that application.

[7] `www.python.org`

An excellent first book for readers unfamiliar with but interested in learning Python is Mark Lutz's *Programming Python* [4].

Finally, the tutorial enlightens us regarding the name:

> ...the language is named after the BBC show *Monty Python's Flying Circus* and has nothing to do with nasty reptiles. Making references to Monty Python skits in documentation is not only allowed, it is encouraged!

7 Combining Python and TEX

There are a number of ways in which to combine TEX and Python to automatically produce documents. If one considers the amount of "work" necessary to produce a document as fixed, then that work can be allocated partly to TEX and partly to Python. One can then imagine a scatter diagram with X and Y axes that represent, for any possible scheme, the amount of work allocated to Python and TEX, respectively. Such a diagram is illustrated in fig. 1.

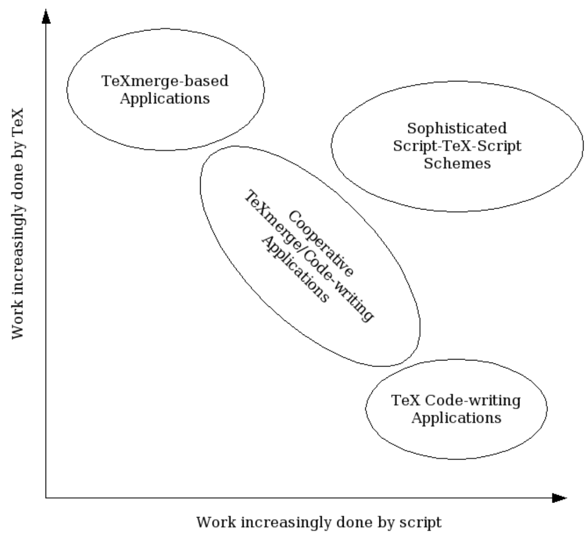

TeXmerge-based Applications
Sophisticated Script-TeX-Script Schemes
Cooperative TeXmerge/Code-writing Applications
TeX Code-writing Applications
Work increasingly done by TeX
Work increasingly done by script

Figure 1: Application Domains of Python/TEX integration.

The diagram shows several different "application domains", defined by which component (TEX or Python) receives the most development effort, or places the most demands on computing resources. These domains allow us to classify various approaches to Python/TEX integration.

7.1 The Imperative Approach

Imagine writing a Python script that produces a file of TEX code by executing a series of `write` statements as in fig. 2 and then runs TEX and `dvips` on that file. Here the emphasis is clearly all on the Python script and the details of how the TEX code is to be produced; we know TEX will dutifully do its job if it is provided good code. Applications of this nature we call *imperative*, and occupy the lower right region of fig. 1.

Figure 2: Imperative TEX code-writing script.

```
#!/usr/bin/env python
import sys
import os
f = open('MyDocument.tex', 'w')
f.write('\\nopagenumbers\n')
f.write('This is my first \\TeX\\ document \
    produced from a script.\n')
f.write('\\vfil\\eject\\bye\n')
f.close()
os.system('tex MyDocument.tex')
os.system('dvips MyDocument')
print 'Done.'
```

This technique is the simplest way to integrate Python and TEX,[8] and is surprisingly effective. Although the example in fig. 2 is trivial, the imperative technique can be used in applications where documents are assembled from a large *database* of text "snippets". Logic in the Python script provides the "smarts" that determine what snippets to select and how to arrange them for presentation to TEX. More logic and scripts of increasing complexity push the application further to the right on the X-axis in fig. 1.

7.2 Using m4

A slight increase in sophistication (but still remaining near the X-axis of fig. 1, is to employ the macro processor program, m4.[9] m4 [8] is an elaborate search-and-replace engine for text. For example, given the text:

```
Hello, NAME, today is DATE.
```

If we present that text to m4 as input with the following command-line:

```
m4 -DNAME=Sally -DDATE='22-June-2004'
```

the output from m4 would appear as:

```
Hello, Sally, today is 22-June-2004.
```

Now we can play the same game as in the imperative approach, but with a new wrinkle: tags can be embedded in our text snippets. Once the TEX code is assembled, it is preprocessed through m4 and *then* presented to TEX. Here are the steps:

1. Assemble TEX code from snippets of text,
2. Gather data for tag-replacement from a data source,
3. Build m4 command line with `-Dname=value` arguments for each unique tag in the TEX file,
4. Execute the command just built and save the output,
5. Present the saved output to TEX.

8 TEXmerge

We now move away from the X-axis of fig. 1.

The m4 approach introduced an important concept: the idea of *template* files. There exist a large class of applications whose function is to produce, for lack of a better term, "form letters".[10] The m4 technique of the previous section lends itself precisely to this *merging* application: Build a `.tex` file with tag names, then repeat steps 2–5 above until end of data. The end result will be a stack of form letters ready to print and drop in the mail.

While m4 is an efficient macro-replacement engine, we know of another engine that eclipses it: TEX. Consider the TEX document in fig. 3.

Figure 3: `form.tex`: A merge-ready TEX file.

```
\nopagenumbers
This is my first \TeX\ document produced
from a script.
\par
Hello, \NAME, today is \DATE.
\vfil\eject
```

Alone, this file will result in undefined macro references because the macros `\NAME` and `\DATE` are not

[8] The other simple extreme would be to prepare an entire document by hand-editing and then have Python run TEX on that file. Quite uninteresting.

[9] Quoting from the m4 manual page: "The m4 utility is a macro processor that can be used as a front end to any language (e.g., C, ratfor, fortran, lex, yacc) ..." and now, TEX!

[10] Every technological advance seems to bring with it a raft of nastiness. With email comes spam, with computer-aided printing comes the dreaded form letter. At least with TEXmerge, they can be beautiful form letters.

defined. However, when used in conjunction with the Python script in fig. 4, it works beautifully.

Figure 4: Imperative TEX code-writing script relying on TEX's macro replacement facility.

```
#!/usr/bin/env python
import sys
import os
f = open('temp.tex', 'w')
f.write('\\def\\NAME{Sally}\n')
f.write('\\def\\DATE{22-June-2004}\n')
f.write('\\input form.tex\n')
f.write('\\bye\n')
f.close()
os.system('tex temp.tex')
os.system('dvips temp')
print 'Done.'
```

Scripts like 4 can be represented schematically as in fig. 5. It is important to note that in this scheme we are dealing with *two* (or more) `.tex` files: 1) The template file(s) containing the structure of our form letter(s) (more than a single type of form letter can be produced in a single run simply by inputting different template files), which have tags where merge variables are to be inserted, and 2) the temporary file which defines macros for the merge variables and has input commands to bring in the templates. Inside the temporary `.tex` file there can be many occurrences of the `\def`... and `\input`... lines; one occurrence for each letter to be produced.

8.1 TEXmerge API

The technique illustrated in fig. 4 works well. Data for the merge variables can be arbitrarily long, for example, and TEX will 'do the right thing' and wrap the merged text into our form, etc. But there are problems:

1. The biggest problem is data containing tokens having special meaning to TEX. If our merge data contains `$`,`%`,`&`, etc., we have a problem,
2. It's rather tedious to read the script, and we find ourselves repeatedly re-implementing this tedious code for every application.

The whole process of opening the temporary `.tex` file, protecting sensitive tokens, preparing the `\def` lines for the merge variables, doing the `\input`..., executing TEX and the DVI backend need to be formalized inside an application programming interface (API).

We call that API "TEXmerge". It was first presented [7] as a C-language API with a Python extension wrapper module. Since that time, the API has been re-written in pure Python and is presented here (see appendix A for a technical description of the API).

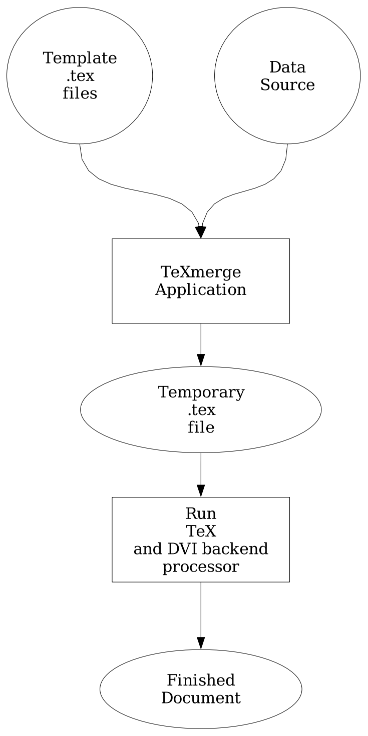


Figure 5: Schematic overview of document production via the TEXmerge API.

First, an example using the TEXmerge API (the TEXmerge *module*). Fig. 6 re-implements the script presented in fig. 4 using the module-level interface:

Figure 6: A simple Python script using the TEXmerge module-level API functions.

```
#!/usr/bin/env python
import sys
import os
import TeXmerge
f = TeXmerge.openOutput('temp.tex')
mergeVars = {'NAME': 'Sally',
             'DATE': '22-June-2004'}
TeXmerge.merge('form.tex', mergeVars)
TeXmerge.closeOutput(f)
TeXmerge.process('temp.tex', 'dvips')
print 'Done.'
```

Note the following:

1. Access to the TEXmerge module is provided via the import statement: `import TeXmerge`.

2. The native Python `open/close` calls have been replaced with calls to `TeXmerge.openOutput()` and `TeXmerge.closeOutput()`.
3. Merge variables are formally presented to the API as a Python dictionary object.
4. The `merge()` call takes care of protecting sensitive tokens in the merge data that would otherwise confuse TEX.
5. The `os.system()` calls have been replaced with `TeXmerge.process()`.

Finally, Python is an object-oriented language, so the TEXmerge module also offers a TEXmerge *class*. Fig. 7 re-implements fig. 6 using the object-oriented interface:

Figure 7: A simple Python script using the TEXmerge object-oriented interface.

```
#!/usr/bin/env python
import sys
import os
import TeXmerge
mergeObj = TeXmerge.TeXmerge('temp.tex')
mergeVars = {'NAME': 'Sally',
             'DATE': '22-June-2004'}
mergeObj.merge('form.tex', mergeVars)
mergeObj.process('dvips')
print 'Done.'
```

9 Going Further with Macros

Now it is time to move up the Y-axis of fig. 1, focus attention on the TEX domain and investigate what benefits can be gained by writing specialized macros to enhance integration with TEXmerge.

9.1 Do-Nothing Macros

The first class of macros to be considered is the "do-nothing" macros. These macros, from TEX's view, evaluate to `\relax`. They exist in a TEXmerge template file in order to communicate information to a Python script which scans the template file. A more traditional method used to communicate information to an external entity would be to embed that information in comment strings within the file. Writing first-class macros, however, seems to produce a cleaner, readable file, and is more flexible since a do-nothing macro could, in the future, be turned into a "*do-something*" macro.

9.1.1 Classic Merge Variable Declarations

Do-nothing macros were introduced in the first release of TEXmerge, with the `\texmergevar` macro. Just looking at a merge-ready template `.tex` file, it is not immediately clear what the names of all the merge variables are. `\texmergevar` allows the author of the template file to explicitly state the names of all merge variables that will be referenced in the file by coding:

`\texmergevar` *name*

for each merge variable. The TEXmerge module has a module-level method, `getNames`, which scans a passed `.tex` file name (and recursively any included files) and returns a list of all declared variable names. Python scripts can inspect TEX template files and determine the names of all declared merge variables.

9.1.2 Extended Merge Variable Declarations

Several years' use of the TEXmerge API has shown that document-producing applications could be made more robust if a template `.tex` file could specify precisely what *values* a merge variable should contain. The need for merge variables to take on only one value from a small set of possible values stems from the use of conditional TEX code, via the `\ifx` control sequence, etc. Conditional typesetting is powerful because it allows documents to become intelligent. *A single* `.tex` *source file can produce entirely different finished documents by testing the value of merge variable(s) and typesetting text accordingly.*

A life insurance company, for example, falls under the jurisdiction of every state in which it is licensed to conduct business. A document, say a "sales practice guide", often must contain language mandated by a particular state. Sales practice guides for forty different states may have 90% of their language in common, but each may also have unique state-specific language that none of the others contains. Having a single, intelligent source file, `salesPracticeGuide.tex`, lowers the cost of change management substantially; changes made to shared text need only be made *once*.

The do-nothing macro `\texmergevardef` defines merge variables with extended attributes, like this:

`\texmergevardef[`*attrName*=*attrValue*,...`]`

Attributes of the merge variables that can be specified are:

- `name` = the name of the merge field.
- `type` = the type of merge field. The intended use of this attribute is to convey a recommended style of data entry element for graphical (GUI) applications. Valid types are:
  - `entry`: a simple text entry field,

Figure 8: A sampling of extended merge variable declarations.

```
\texmergevardef[name=ISTATE, type=optionmenu,values=TX|OK|AZ|CA|OR|WA,descr=Issuing state]
\texmergevardef[name=ONAME,type=entry,descr=Owner name]
\texmergevardef[name=APPTYPE,type=radiobutton,values=1|2|3,labels=Employee|Spouse|Child,
                descr=Applicant type]
```

Figure 9: Result of `getExtendedNames()`: a Python dictionary of field-attribute dictionaries

```
{'ISTATE': {'name': 'ISTATE', 'type': 'optionmenu', 'values': ('TX', 'OK', 'AZ', 'CA', 'OR', 'WA'),
'descr' : 'Issuing state'}, 'APPTYPE': {'name': 'APPTYPE', 'type': 'radiobutton','values':
('1', '2', '3'), 'labels': ('Employee', 'Spouse', 'Child'), 'descr': 'Applicant type'}, 'ONAME: {
'name': 'ONAME', 'type': 'entry', 'descr': 'Owner name'}}
```

  - `text`: a multi-line text entry field,
  - `toggle`: a toggle button field,
  - `optionmenu`: a drop-down option menu of choices,
  - `radiobutton`: a set of mutually-exclusive toggle buttons.
- `values` = a list of valid values for the variable, separated by `|`'s.
- `labels` = a list of alternate labels that should be associated with the `values` attribute for display purposes. Used with the `toggle`, `optionmenu`, and `radiobutton` field types.
- `descr` = a description of the merge variable.

The TEXmerge module-level function `getExtendedNames` extracts these extended merge variable definitions, parses them, and returns them in a dictionary (keyed by the `name` attribute's value) of field attribute dictionaries.[11] Fig. 8 shows an example `.tex` file with extended merge variable definitions. Fig. 9 shows the return value from applying `getExtendedNames` on that file.

9.1.3 Named Text Blocks

Another class of applications has the need to share identical text between two markup languages: TEX and HTML. Here it is *language within the document* that needs to be identical (for legal reasons, say) and not the structure of the document that is constant between the two presentation platforms. Indeed, structure of the printed TEX document may be substantially more complex than its briefer, lightweight, HTML cousin. How can the common text be shared between the markup languages?

One way is to make the TEX document "own" the text. It declares, via a set of macros, where the common blocks of text begin and end. We refer to these blocks as *named text blocks*. The demarcation macros look like this:

- `\StartNamedTextBlock[`*attrName*=*value*...`]`
  Text block attributes are as follows:
  - `name` = Name of the text block,
  - `seq` =*Integer*; several sections of text can be assigned the same name, but with unique sequence numbers. The extracted text will be a concatenation of like-named blocks, ordered by sequence number,
  - *subkey* =*subvalue*: See the text for full discussion.
- `\StopNamedTextBlock`

Once text boundaries have been marked and named with these macros, the text can be extracted and used by the HTML producing part of the application. The TEXmerge module provides a module-level function, `getNamedTextBlocks`, to extract the named text blocks, and two helper classes `TextBlock` and `TextBlockManager` to make accessing the extracted blocks simpler.

We explain the functional use of named text blocks by way of the example file in fig. 10 and the interactive Python interpreter session shown in fig. 11.[12]

Note the following:

1. The block demarcation macros are essentially invisible to TEX and have no effect on typesetting.
2. The `TextBlockManager` class is used to extract the named blocks. One simply passes a pathname to the `.tex` file containing named text

[11] `getExtendedNames` also detects occurrences of the prior `texmergevar` macro and treats them as extended merge fields having an attribute `type` = `entry`.

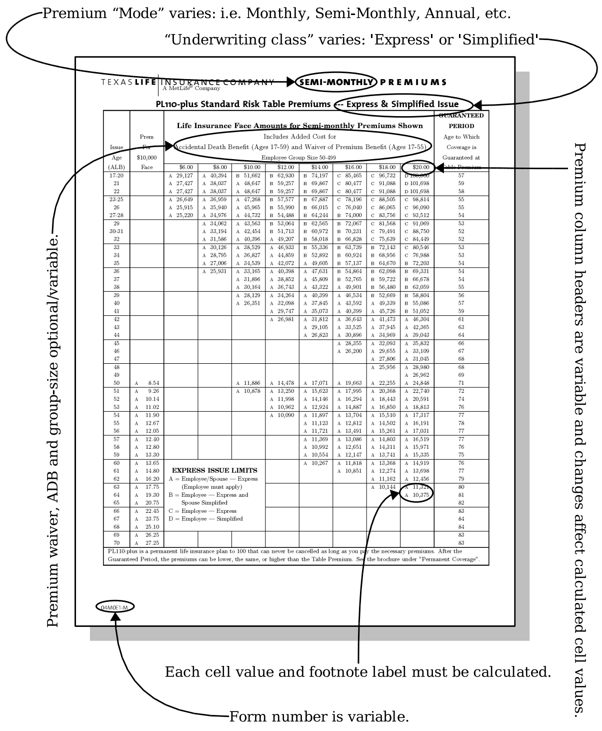
[12] About the interactive interpreter session: >>> is the interpreter's prompt. Text appearing after that prompt was entered by the user. Python's response appears on the line immediately below the prompt input line.

Figure 10: TEX file `test.tex` containing four named text blocks: B1, B2, C1, D1.

```
This is a test document containing \textit{named text blocks.}
\StartNamedTextBlock[name=B1]
This is the first block.
\StopNamedTextBlock
Now for a second block:
\StartNamedTextBlock[name=B2]
Second block
\StopNamedTextBlock
Now for a series of sequenced blocks \ldots
\line{\hbox{\StartNamedTextBlock[name=C1,seq=1]C1.Left\StopNamedTextBlock}\hfil
      \hbox{\StartNamedTextBlock[name=C1,seq=2]C1.Right\StopNamedTextBlock}
}
Finally, a named text block having a subkey:
\StartNamedTextBlock[name=D1,istate=TX]
This text is specific to the state of Texas.
\StopNamedTextBlock
```

Figure 11: Interactive Python interpreter session. Working with named text blocks.

```
[hawkeye2:~/sftug] williamr% python
Python 2.3.2 (#1, Nov  6 2003, 13:18:07)
[GCC 2.95.2 19991024 (release)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import TeXmerge
>>> o = TeXmerge.TextBlockManager('test.tex')
>>> o
<TeXmerge.TextBlockManager instance at 0x750648>
>>> o.getBlockNames()
['C1', 'B1', 'B2', 'D1']
>>> b1 = o.getBlock('B1')
>>> b1
<TeXmerge.TextBlock instance at 0x72b5d0>
>>> b1.getText()
'This is the first block.'
>>> c1 = o['C1']
>>> c1.getTextSegments()
{1: 'C1.Left', 2: 'C1.Right'}
>>> c1.getText()
'C1.Left C1.Right'
>>> d1 = o['D1']
>>> d1.getSubkeys()
['istate']
>>> d1.getSubkeyValues('istate')
['TX']
>>> d1.getText('istate','TX')
'This text is specific to the state of Texas.'
```


Premium "Mode" varies: i.e. Monthly, Semi-Monthly, Annual, etc.
"Underwriting class" varies: 'Express' or 'Simplified'
Premium waiver, ADB and group-size optional/variable.
Premium column headers are variable and changes affect calculated cell values.
Each cell value and footnote label must be calculated.
Form number is variable.

TEXAS LIFE INSURANCE COMPANY
A MetLife® Company

SEMI-MONTHLY PREMIUMS

PL110-plus Standard Risk Table Premiums --- Express & Simplified Issue

Issue Age (ALB)	Prem For $10,000 Face	Life Insurance Face Amounts for Semi-monthly Premiums Shown Includes Added Cost for Accidental Death Benefit (Ages 17-59) and Waiver of Premium Benefit (Ages 17-55) Employee Group Size 50-499								GUARANTEED PERIOD Age to Which Coverage is Guaranteed at Table Premium
		$6.00	$8.00	$10.00	$12.00	$14.00	$16.00	$18.00	$20.00	
17-20		A 29,127	A 40,394	B 51,662	B 62,930	B 74,197	C 85,465	C 96,732	D 100,000	57
21		A 27,427	A 38,037	A 48,647	B 59,257	B 69,867	C 80,477	C 91,088	D 101,698	59
22		A 27,427	A 38,037	A 48,647	B 59,257	B 69,867	C 80,477	C 91,088	D 101,698	58
23-25		A 26,649	A 36,959	A 47,268	B 57,577	B 67,887	C 78,196	C 88,505	C 98,814	55
26		A 25,915	A 35,940	A 45,965	B 55,990	B 66,015	C 76,040	C 86,065	C 96,090	55
27-28		A 25,220	A 34,976	A 44,732	B 54,488	B 64,244	B 74,000	C 83,756	C 93,512	54
29			A 34,062	A 43,563	B 53,064	B 62,565	B 72,067	C 81,568	C 91,069	53
30-31			A 33,194	A 42,454	B 51,713	B 60,972	B 70,231	C 79,491	C 88,750	52
32			A 31,586	A 40,396	A 49,207	B 58,018	B 66,828	C 75,639	C 84,449	52
33			A 30,126	A 38,529	A 46,933	B 55,336	B 63,739	B 72,143	C 80,546	53
34			A 28,795	A 36,827	A 44,859	B 52,892	B 60,924	B 68,956	C 76,988	53
35			A 27,006	A 34,539	A 42,072	A 49,605	B 57,137	B 64,670	B 72,203	54
36			A 25,931	A 33,165	A 40,398	A 47,631	B 54,864	B 62,098	B 69,331	54
37				A 31,896	A 38,852	A 45,809	B 52,765	B 59,722	B 66,678	54
38				A 30,164	A 36,743	A 43,322	A 49,901	B 56,480	B 63,059	55
39				A 28,129	A 34,264	A 40,399	A 46,534	B 52,669	B 58,804	56
40				A 26,351	A 32,098	A 37,845	A 43,592	A 49,339	B 55,086	57
41					A 29,747	A 35,073	A 40,399	A 45,726	B 51,052	59
42					A 26,981	A 31,812	A 36,643	A 41,473	A 46,304	61
43						A 29,105	A 33,525	A 37,945	A 42,365	63
44						A 26,823	A 30,896	A 34,969	A 39,043	64
45							A 28,355	A 32,093	A 35,832	66
46							A 26,200	A 29,655	A 33,109	67
47								A 27,806	A 31,045	68
48								A 25,956	A 28,980	68
49									A 26,962	69
50	A 8.54			A 11,886	A 14,478	A 17,071	A 19,663	A 22,255	A 24,848	71
51	A 9.26			A 10,878	A 13,250	A 15,623	A 17,995	A 20,368	A 22,740	72
52	A 10.14				A 11,998	A 14,146	A 16,294	A 18,443	A 20,591	74
53	A 11.02				A 10,962	A 12,924	A 14,887	A 16,850	A 18,813	76
54	A 11.90				A 10,090	A 11,897	A 13,704	A 15,510	A 17,317	77
55	A 12.67					A 11,123	A 12,812	A 14,502	A 16,191	78
56	A 12.05					A 11,721	A 13,491	A 15,261	A 17,031	77
57	A 12.40					A 11,369	A 13,086	A 14,803	A 16,519	77
58	A 12.80					A 10,992	A 12,651	A 14,311	A 15,971	76
59	A 13.30					A 10,554	A 12,147	A 13,741	A 15,335	75
60	A 13.65					A 10,267	A 11,818	A 13,368	A 14,919	76
61	A 14.80						A 10,851	A 12,274	A 13,698	77
62	A 16.20							A 11,162	A 12,456	79
63	A 17.75							A 10,144	A 11,321	80
64	A 19.30								A 10,375	81
65	A 20.75									82
66	A 22.45									83
67	A 23.75									84
68	A 25.10									84
69	A 26.25									83
70	A 27.25									83

EXPRESS ISSUE LIMITS
A = Employee/Spouse — Express (Employee must apply)
B = Employee — Express and Spouse Simplified
C = Employee — Express
D = Employee — Simplified

PL110-plus is a permanent life insurance plan to 100 that can never be cancelled as long as you pay the necessary premiums. After the Guaranteed Period, the premiums can be lower, the same, or higher than the Table Premium. See the brochure under "Permanent Coverage".

04M0E1-M

Figure 12: Complex document produced by Hybrid Script-TEX-Script scheme.

blocks in order to instantiate an object of the `TextBlockManager` class.

3. The names of all the text blocks in the file can be retrieved by calling the manager object's `getNamedTextBlocks` method.
4. Individually named text blocks are retrieved via the manager object's `getTextBlock` method, or simply by indexing the manager using the name of a text block as the index key (as was done for block C1 in fig. 11). Either operation will return a `TextBlock` object.
5. Access to the text of a `TextBlock` object is via its `getText` method.

9.2 Do-Something Macros

9.2.1 Hybrid Script-TEX-Script Scheme: A Case Study

If we have an application where a substantial amount of the document's content may vary, the merge paradigm of TEXmerge begins to break down under the complexity of so many variables. This is especially true of variable tabular data.

Example: The annotated page shown in fig. 12 is a rate sheet of life insurance premiums. As the figure shows, there is more variable data than static text on the page. The rate sheet, however, is only one page of a twenty page document. Other pages in its parent document also have variable data, and state-specific language, as well. Overall the document's nature fits well in the TEXmerge scheme; the rate sheet page is the "trouble maker". Another important consideration: the rate sheet needs to be embeddable in many other documents.

We desire a TEX macro as in fig. 13 that, when executed, magically produces a finished rate sheet.[13]

Figure 13: Rate sheet macro.

```
\MakeRateSheet[uwclass=express,
               mode=semi-monthly,
               groupsize=150,
               formno=test,
               waiver=yes,
               adb=yes
]
```

`\MakeRateSheet[...]` is definitely a *do-something* macro. The trick is to do as little work as possible in TEX and most of the *something* in a Python script. The work for TEX in this case is in two parts:

1. Gather macro arguments and marshal them into a Python script command-line, then execute the command with `\write18`.
2. Input and typeset the TEX code produced by the Python script.

We call schemes such as these *hybrid* or *Script-TEX-Script* schemes. The job of the secondary script (the one executed by TEX via `\write18`) is to act on arguments received from TEX, or from some other external source, do whatever calculations, etc., and output TEX code. The whole scheme is represented in fig. 14. Since the secondary script is unbounded by the complexity and amount of TEX code that may be returned, hybrid schemes are the ultimate in flexibility.

9.2.2 Document Template Macros

Document template macros also fall into the class of *do-something* macros. Another case study will serve as a description of their functionality. TEXmerge is in widespread use at Texas Life having applications in almost every major department, from Marketing, to New Business, to Policy Owner Service, to Computing Services. Several years ago, a graphic artist was hired to develop a new 'look-and-feel' for all printed material disseminated from the company. A

[13] Writing parameter based macros such as these is effortless with the aid of support macros found in Hans Hagen's ConTEXt macro package.

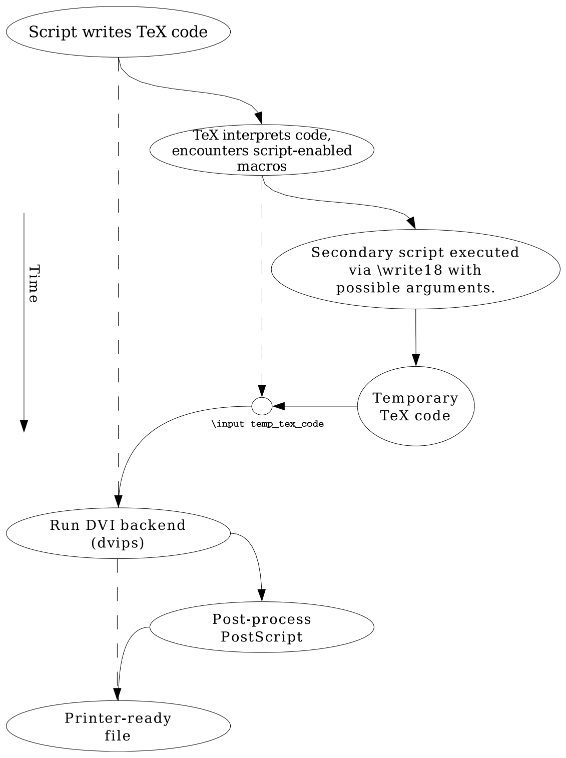


Figure 14: Schematic overview of document production via the hybrid technique.

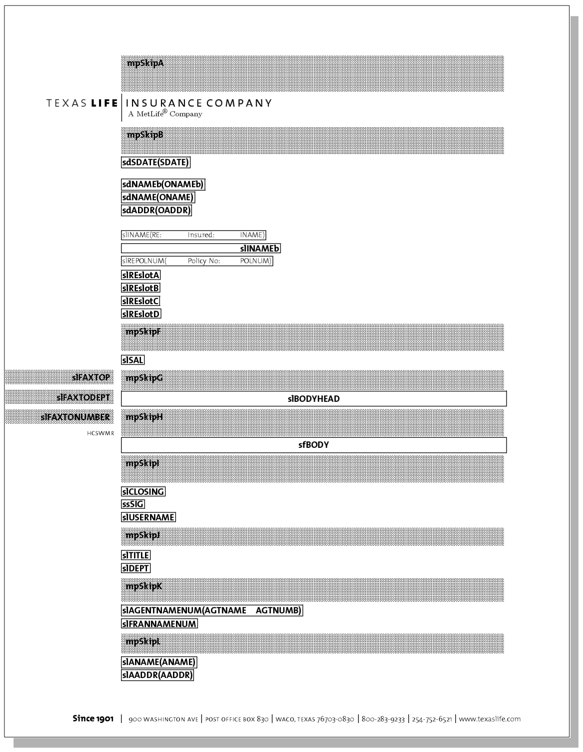


Figure 15: Template view for the client-letter macro.

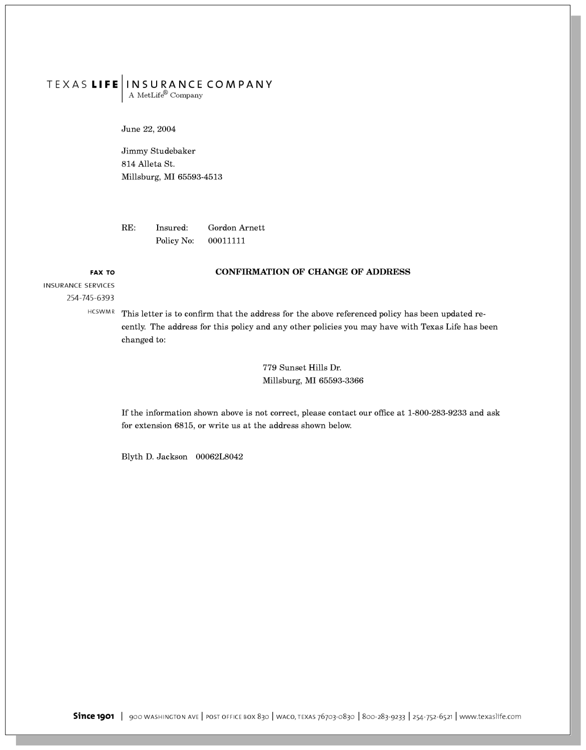
new graphics standards manual was written and all parts of the company were informed that compliance with the new standard was mandatory by a set date. This directly affected users of TEXmerge. The Policy Owner Service department, for example, had 600+ TEXmerge-based form letters used daily for corresponding with clients. Compounding the problem were the non-standard fonts and a peculiar format to which standard letterhead should conform: a wide left margin, except for various items that were to remain left hanging, right-justified. How could over 600 documents be quickly converted to this new format? Language inside the documents could remain unaltered; only the structure was changing.

Serendipitous earlier decisions, made when originally planning and setting up the TEXmerge letters, made conversion to the new graphics standard straightforward. The serendipity was in a decision to separate the text for the body of each letter into its own `.tex` file. That being the case, all that was needed was a mechanism to enforce the policy of the graphics standard; a way to automatically produce the required layout of the document. This we do with so-called *template* macros. Fig. 15 shows the structure enforced by the `\StartClientLetter` macro. Based on a *plug-and-socket* model, it relies heavily on macro parameters (almost all having default values), as can be seen in the figure. Template macros classify parameters into three categories:

- Simple parameters: parameter names beginning with `mp`,
- Data sockets: parameter names beginning with `sd`,
- Slots: parameter names beginning with `sl`.[14]

The `mpSkip...` parameters (gray strips shown in fig. 15) can be specified to alter whitespace. Merge variable data is connected to a template using a *plug-and-socket* model. Merge variable names are termed *plugs* and the `sd...` macro parameters are termed sockets. One plugs a variable to particular position on the letter by equating the name of the plug with the desired socket name. The socket names are shown on the template letter in fig. 15 with default plug values in parenthesis. Finally, *slots* are macro parameters that can accept arbitrary TEX code as arguments.

[14] There are two other prefixes: `ss`, related to insertion of digitized versions of handwriting signatures; and `sf`, related to input files.



TEXAS **LIFE** | INSURANCE COMPANY
A MetLife® Company

June 22, 2004

Jimmy Studebaker
814 Alleta St.
Millsburg, MI 65593-4513

RE: Insured: Gordon Arnett
Policy No: 00011111

FAX TO
INSURANCE SERVICES
254-745-6393
HCSWMR

CONFIRMATION OF CHANGE OF ADDRESS

This letter is to confirm that the address for the above referenced policy has been updated recently. The address for this policy and any other policies you may have with Texas Life has been changed to:

779 Sunset Hills Dr.
Millsburg, MI 65593-3366

If the information shown above is not correct, please contact our office at 1-800-283-9233 and ask for extension 6815, or write us at the address shown below.

Blyth D. Jackson 00062L8042

Since 1901 | 900 WASHINGTON AVE | POST OFFICE BOX 830 | WACO, TEXAS 76703-0830 | 800-283-9233 | 254-752-6521 | www.texaslife.com

Figure 16: Sample letter produced using the client-letter macro.

The body of the letter can be supplied to the template macro in one of two ways:

1. Put the text of the body into a separate `.tex` file and pass the name of the file in the `sfBODY` parameter,
2. Code text of the body immediately after invoking `\StartClientLetter`. In this case the letter must end with `\FinishClientLetter`.

Finally, fig. 16 shows a sample letter produced from the `\StartClientLetter` macro.

10 Building GUI Applications with TEXmerge

So far, our discussion of TEXmerge has tended toward batch-style applications. The API is also effective in building GUI applications. The module's `getNames` and `getExtendedNames` functions provide useful *metadata* about merge fields, which can be used to construct user interfaces. Python is equally effective in programming GUI interfaces. The "Gimp Toolkit"[15] is especially easy to access from Python and provides a robust set of GUI interface components, including Pixmap buffers which, along with Ghostscript[16], can be used to effectively render PostScript.

[15] `www.gtk.org` and `www.pygtk.org`.

10.1 TEXmerge — the Application

The TEXmerge API was originally developed for use in an interactive application, also called TEXmerge, for production of form letters. Originally written in C and based on the Motif toolkit, the current version is written in pure Python and is based on GTK+ 2.4. The application is arranged around *categories* of correspondence (collections of form letters, grouped by activity. Each activity category's letters are stored in a category subdirectory.

A sample TEXmerge main application window is shown in fig. 17. A category frame consists of the document selection window on the left, and a set of merge variable data entry fields on the right. A single set of input fields (a *record*), generates a single copy of the associated letter. Control buttons exist along the bottom to accomplish tasks such as adding new records, removing records, printing, and saving. A built-in PostScript viewer (not visible) is also provided to view the letter before printing or saving.

10.2 TEXtool

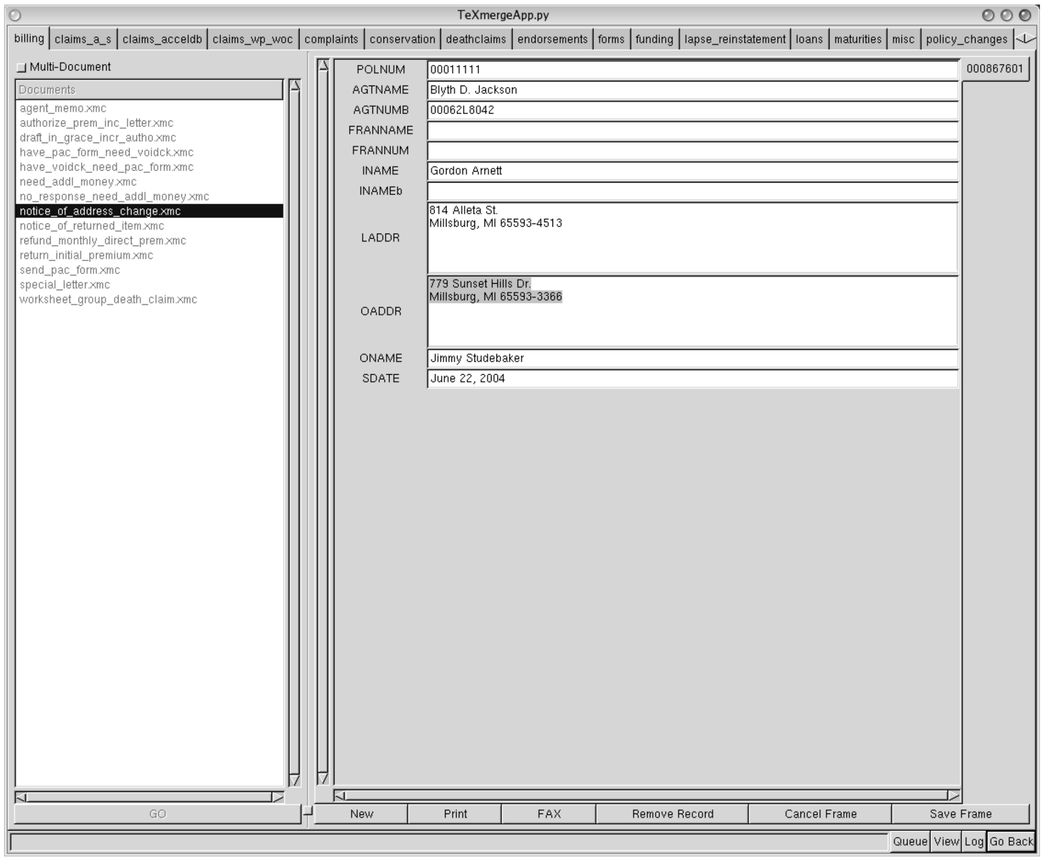
As long as we're writing GUI applications, why not write one that aids in the development of TEXmerge documents? TEXtool is an integrated development utility for editing, "TEX'ing", and viewing TEXmerge documents. Figs. 18, 19, and 20 are three successive views of the application, each view revealing one of the major notebook tab pages: Document, Editor, and Preferences.

Applications of this style exist that are more effective in general; however, TEXtool is unique because it is oriented especially for TEXmerge documents. It also shows the feasibility of integrating TEX into a non-trivial GUI application written in a scripting language. As can been gleaned from the figures, the Document tab displays the input frame of TEXmerge variables as they will appear in the normal TEXmerge application. The edit/test cycle can be quickly done all inside a single application window.

11 The Big Picture at Texas Life

As mentioned in the case studies earlier, TEXmerge is in widespread use at Texas Life. Fig. 21 is reproduced from [7]. It is a convincing illustration of how effective TEX can be as a document production engine, especially if combined with the right scripting

[16] `www.ghostscript.com`



TeXmergeApp.py

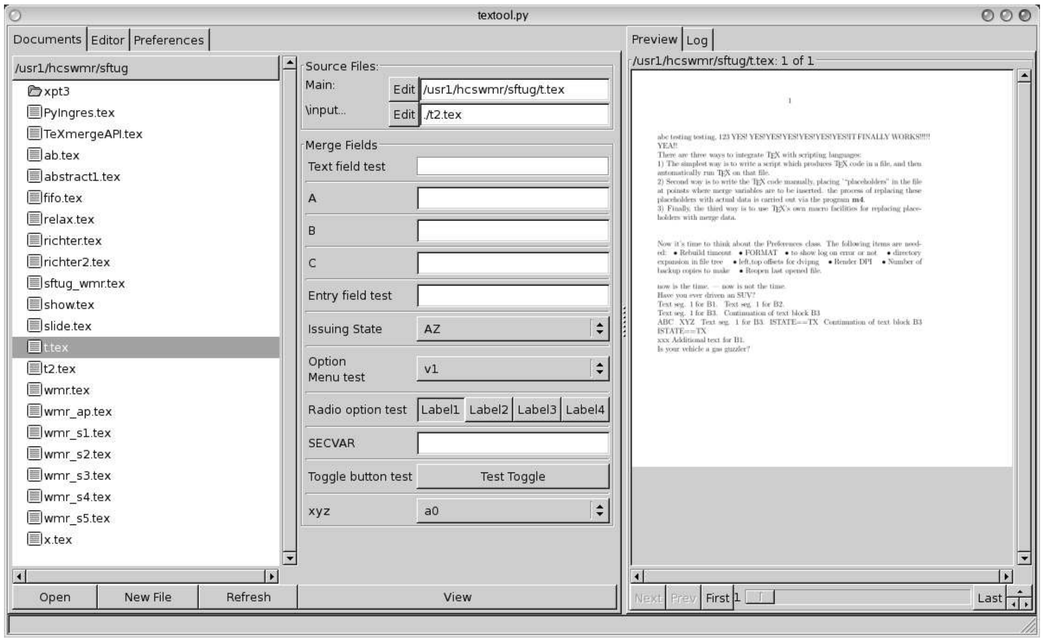
Multi-Document

Documents
- agent_memo.xmc
- authorize_prem_inc_letter.xmc
- draft_in_grace_incr_autho.xmc
- have_pac_form_need_voidck.xmc
- have_voidck_need_pac_form.xmc
- need_addl_money.xmc
- no_response_need_addl_money.xmc
- notice_of_address_change.xmc
- notice_of_returned_item.xmc
- refund_monthly_direct_prem.xmc
- return_initial_premium.xmc
- send_pac_form.xmc
- special_letter.xmc
- worksheet_group_death_claim.xmc

POLNUM	00011111
AGTNAME	Blyth D. Jackson
AGTNUMB	00062L8042
FRANNAME	
FRANNUM	
INAME	Gordon Arnett
INAMEb	
LADDR	814 Alleta St. Millsburg, MI 65593-4513
OADDR	779 Sunset Hills Dr. Millsburg, MI 65593-3366
ONAME	Jimmy Studebaker
SDATE	June 22, 2004

000867601

Figure 17: The TEXmerge application main window.



textool.py

/usr1/hcswmr/sftug
- xpt3
- PyIngres.tex
- TeXmergeAPI.tex
- ab.tex
- abstract1.tex
- fifo.tex
- relax.tex
- richter.tex
- richter2.tex
- sftug_wmr.tex
- show.tex
- slide.tex
- t.tex
- t2.tex
- wmr.tex
- wmr_ap.tex
- wmr_s1.tex
- wmr_s2.tex
- wmr_s3.tex
- wmr_s4.tex
- wmr_s5.tex
- x.tex

Source Files:

Main:	/usr1/hcswmr/sftug/t.tex
\input...	./t2.tex

Merge Fields

Text field test	
A	
B	
C	
Entry field test	
Issuing State	AZ
Option Menu test	v1
Radio option test	Label1 Label2 Label3 Label4
SECVAR	
Toggle button test	Test Toggle
xyz	a0

/usr1/hcswmr/sftug/t.tex: 1 of 1

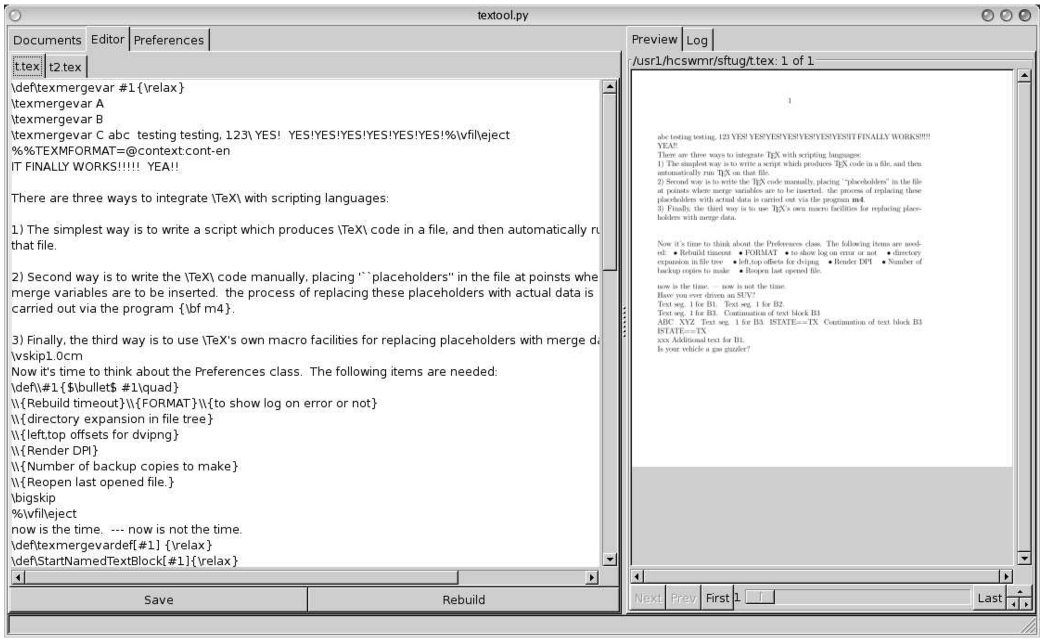
1

abc testing testing, 123 YES! YES!YES!YES!YES!YES!YES!IT FINALLY WORKS!!!!!
YEA!!
There are three ways to integrate TEX with scripting languages:
1) The simplest way is to write a script which produces TEX code in a file, and then automatically run TEX on that file.
2) Second way is to write the TEX code manually, placing "placeholders" in the file at points where merge variables are to be inserted. the process of replacing these placeholders with actual data is carried out via the program **m4**.
3) Finally, the third way is to use TEX's own macro facilities for replacing placeholders with merge data.

Now it's time to think about the Preferences class. The following items are needed: • Rebuild timeout • FORMAT • to show log on error or not • directory expansion in file tree • left,top offsets for dvipng • Render DPI • Number of backup copies to make • Reopen last opened file.

now is the time. — now is not the time.
Have you ever driven an SUV?
Text seg. 1 for B1. Text seg. 1 for B2.
Text seg. 1 for B3. Continuation of text block B3
ABC XYZ Text seg. 1 for B3. ISTATE==TX Continuation of text block B3
ISTATE==TX
xxx Additional text for B1.
Is your vehicle a gas guzzler?

Figure 18: The `textool` application with the Documents tab visible.



```
\def\texmergevar #1{\relax}
\texmergevar A
\texmergevar B
\texmergevar C abc  testing testing, 123\ YES!  YES!YES!YES!YES!YES!YES!%\vfil\eject
%%TEXMFORMAT=@context:cont-en
IT FINALLY WORKS!!!!!  YEA!!

There are three ways to integrate \TeX\ with scripting languages:

1) The simplest way is to write a script which produces \TeX\ code in a file, and then automatically ru
that file.

2) Second way is to write the \TeX\ code manually, placing '``placeholders'' in the file at poinsts whe
merge variables are to be inserted.  the process of replacing these placeholders with actual data is
carried out via the program {\bf m4}.

3) Finally, the third way is to use \TeX's own macro facilities for replacing placeholders with merge da
\vskip1.0cm
Now it's time to think about the Preferences class.  The following items are needed:
\def\\#1{$\bullet$ #1\quad}
\\{Rebuild timeout}\\{FORMAT}\\{to show log on error or not}
\\{directory expansion in file tree}
\\{left,top offsets for dvipng}
\\{Render DPI}
\\{Number of backup copies to make}
\\{Reopen last opened file.}
\bigskip
%\vfil\eject
now is the time.  --- now is not the time.
\def\texmergevardef[#1] {\relax}
\def\StartNamedTextBlock[#1]{\relax}
```

/usr1/hcswmr/sftug/t.tex: 1 of 1

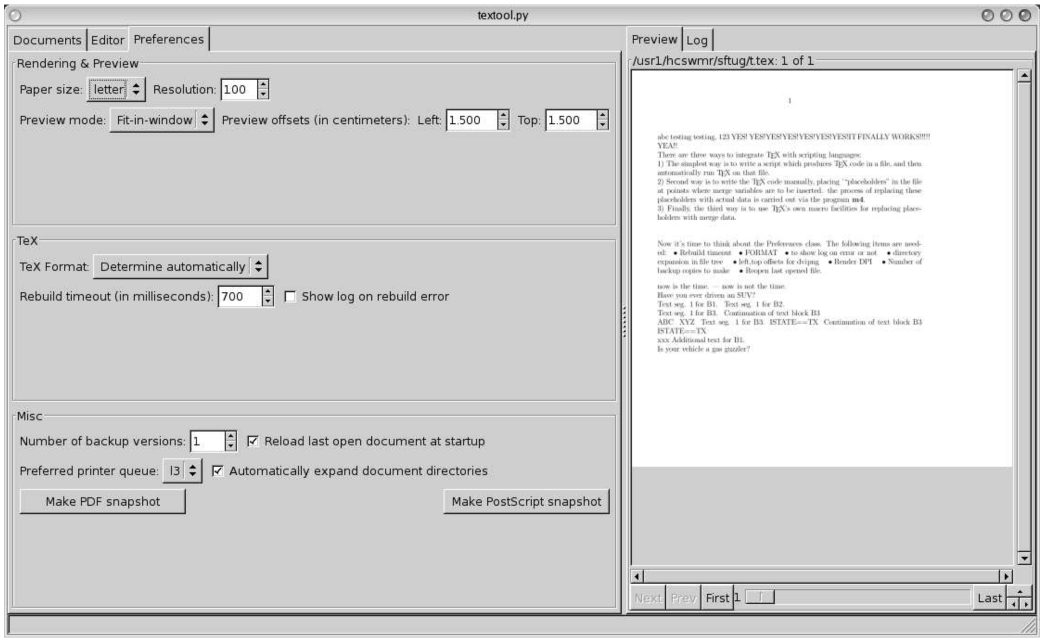
1

abc testing testing, 123 YES! YES!YES!YES!YES!YES!YES!IT FINALLY WORKS!!!!!
YEA!!
There are three ways to integrate TeX with scripting languages:
1) The simplest way is to write a script which produces TeX code in a file, and then automatically run TeX on that file.
2) Second way is to write the TeX code manually, placing '"placeholders" in the file at poinsts where merge variables are to be inserted. the process of replacing these placeholders with actual data is carried out via the program **m4**.
3) Finally, the third way is to use TeX's own macro facilities for replacing placeholders with merge data.

Now it's time to think about the Preferences class. The following items are needed: • Rebuild timeout • FORMAT • to show log on error or not • directory expansion in file tree • left,top offsets for dvipng • Render DPI • Number of backup copies to make • Reopen last opened file.

now is the time. — now is not the time.
Have you ever driven an SUV?
Text seg. 1 for B1. Text seg. 1 for B2.
Text seg. 1 for B3. Continuation of text block B3
ABC XYZ Text seg. 1 for B3. ISTATE==TX Continuation of text block B3
ISTATE==TX
xxx Additional text for B1.
Is your vehicle a gas guzzler?

Figure 19: The `textool` application with the Editor tab visible.



Rendering & Preview

Paper size: letter Resolution: 100

Preview mode: Fit-in-window Preview offsets (in centimeters): Left: 1.500 Top: 1.500

TeX

TeX Format: Determine automatically

Rebuild timeout (in milliseconds): 700 ☐ Show log on rebuild error

Misc

Number of backup versions: 1 ☑ Reload last open document at startup

Preferred printer queue: l3 ☑ Automatically expand document directories

/usr1/hcswmr/sftug/t.tex: 1 of 1

1

abc testing testing, 123 YES! YES!YES!YES!YES!YES!YES!IT FINALLY WORKS!!!!!
YEA!!
There are three ways to integrate TeX with scripting languages:
1) The simplest way is to write a script which produces TeX code in a file, and then automatically run TeX on that file.
2) Second way is to write the TeX code manually, placing '"placeholders" in the file at poinsts where merge variables are to be inserted. the process of replacing these placeholders with actual data is carried out via the program **m4**.
3) Finally, the third way is to use TeX's own macro facilities for replacing placeholders with merge data.

Now it's time to think about the Preferences class. The following items are needed: • Rebuild timeout • FORMAT • to show log on error or not • directory expansion in file tree • left,top offsets for dvipng • Render DPI • Number of backup copies to make • Reopen last opened file.

now is the time. — now is not the time.
Have you ever driven an SUV?
Text seg. 1 for B1. Text seg. 1 for B2.
Text seg. 1 for B3. Continuation of text block B3
ABC XYZ Text seg. 1 for B3. ISTATE==TX Continuation of text block B3
ISTATE==TX
xxx Additional text for B1.
Is your vehicle a gas guzzler?

Figure 20: The `textool` application with the Preferences tab visible.

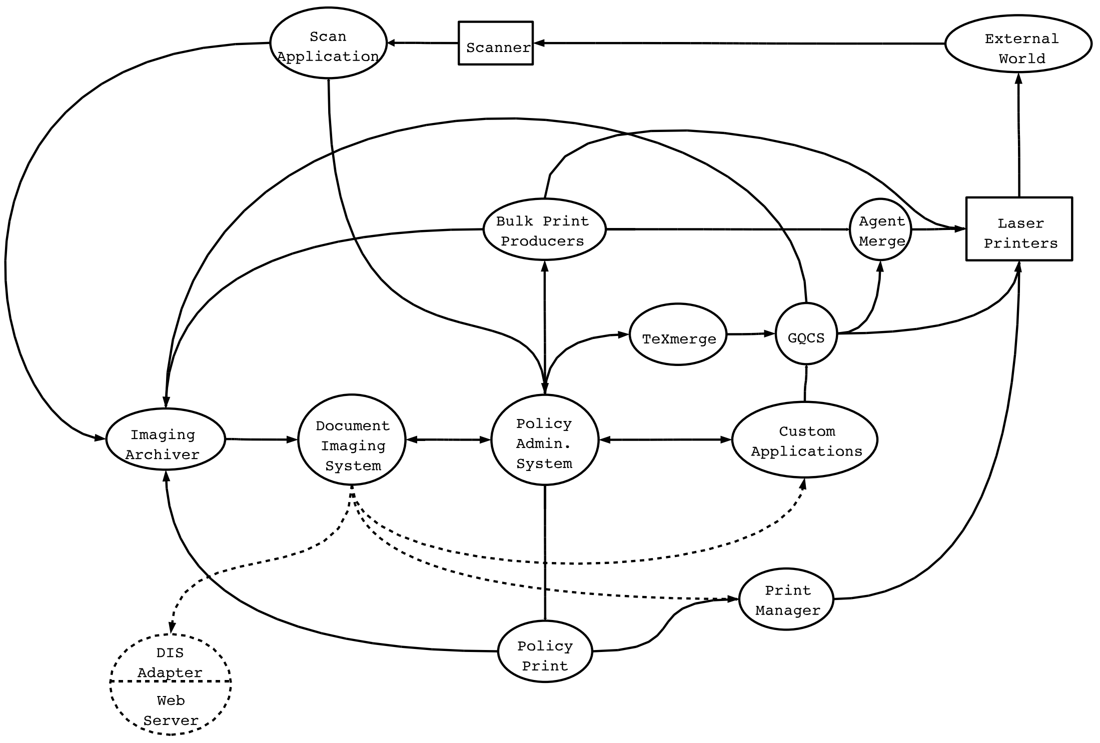


Figure 21: The big picture of TEXmerge at Texas Life.

language (Python). Most of the ovals in the figure use TEXmerge in some fashion. An important lesson learned is that once a facility like TEXmerge is available, the movement of documents between systems becomes much simpler. Only data required to build documents need be communicated along the arrows in the figure. Documents are only built and rendered when necessary for viewing or printing.

12 Conclusion

Because TEX is an ASCII text markup language, it is effective to write computer codes to process the TEX code for purposes other than typesetting. Scripting languages simplify writing these extraction codes. Embedding metadata into TEX files via simple macros allows the TEX author to communicate information to other computer applications. And, finally, using TEX alongside scripting languages in an automated document production environment provides flexibility and robustness to meet almost any demand imaginable. "Hacking" with scripting languages has never been simpler. Now is the time for more people to become *script literate*; the author encourages those with little or no programming experience to mix up a scripting language with their favorite TEX macro package.

References

[1] Bruce Eckel. Strong typing vs. strong testing. 2003. `http://www.mindview.net/WebLog/log-0025`.

[2] Paul Graham. Hackers and painters. 2004. `http://www.paulgraham.com/hp.html`.

[3] Donald E. Knuth. *The Art of Computer Programming*, volume 1. Addison-Wesley, third edition, 1997.

[4] Mark Lutz. *Python Programming*. O'Reilly and Associates, Inc., first edition, 1996.

[5] Eric S. Raymond. The art of Unix programming. 2003. `http://www.faqs.org/doc/artu/ch14s01.html`.

[6] Eric S. Raymond. The meaning of 'hack'. 2003. `http://www.catb.org/~esr/jargon/html/meaning-of-hack.html`.

[7] William M. Richter. Integrating TEX into a document imaging system. *TUGboat*, 22(3), 2001.

[8] René Seindal. GNU m4 development site. 2003. `http://www.seindal.dk/rene/gnu`.

[9] Unknown. Technical definition of scripting language. 2003. `http://c2.com/cgi/wiki?ScriptingLanguage`.

[10] Webopedia. What is object oriented programming? 2003. `http://webopedia.com/TERM/O/object-oriented-programming-OOP.html`.

Appendix A The TEXmerge Python API

A.1 How TEXmerge Runs TEX

Because there are a significant number of macro packages available as TEX formats, TEXmerge needs to be adaptable, both to the format to use, and to the way in which the TEX interpreter is started. To allow for this flexibility, many of the functions below take two arguments, `format` and `strategy`; `format` specifies what TEX format to use and `strategy` specifies the way in which TEX will be started. In many cases, these arguments are optional and appropriate values will be derived, either from the context of use or from the environment variable `TEXMFORMAT`. The environment variable has two different forms:

1. `TEXMFORMAT=`*format*
2. `TEXMFORMAT=@`*strategy*`:`*format*

The second form allows for specification of both the strategy and format. Currently `strategy` can be set to one of: `context`, `latex`,[17] or `plain`. The table below maps strategies to command lines:

strategy	command line
`context`	`texexec --format` *format* `--once %s`
`latex`	`latex %s`
`plain`	`tex &`*format* `%s`

A.2 Module-level Functions

`getNames(`*pathname*`)` → [*name*$_1$, *name*$_2$, . . .]
Recursively scans the passed *pathname* and returns a list of merge variable names declared by instances of the `\texmergevar` macro.

`getExtendedNames(`*pathname*`)` → {*attrDict*$_1$, *attrDict*$_2$, . . . }
Recursively scans the passed *pathname* and returns a dictionary of merge variable field attribute dictionaries. These dictionaries are created from instances of the `\texmergevardef` macro, which defines merge variables with extended attributes, like this:

`\texmergevardef[`*attrName=attrValue,*`...]`

Attributes of the merge variables that can be specified are:

- `name` = the name of the merge field,
- `type` = the type of merge field. The intended use of this attribute is to convey a recommended style of data entry element for graphical (GUI) applications. Valid types are:
  - `entry`: a simple text entry field,
  - `text`: a multi-line text entry field,
  - `toggle`: a toggle button field,
  - `optionmenu`: a drop-down option menu of choices,
  - `radiobutton`: a set of mutually-exclusive toggle buttons
- `values` = a list of valid values for the variable, separated by `|`'s.
- `labels` = a list of alternate labels that should be associated with the `values` attribute for display purposes. Used with the `toggle`, `optionmenu`, and `radiobutton` field types.
- `descr` = a description of the merge variable.

[17] For the `latex` strategy, `format = latex` is always assumed.

`hashNames(`*fieldAttributesDict*`)` → *StringObject*
Computes a 64-bit MD5 hash of the passed field attributes dictionary and returns it as a string object of hexadecimal characters.

`getInputFiles(`*pathname*`)` → [*pathname*$_1$, *pathname*$_2$, . . .]
Recursively scans the passed *pathname* for occurrences of `\input` control sequences and returns a list of pathnames.

`openOutput(`*pathnameOrFileObject*`,` *preambleCode*`=None,` *formatIn*`=None,` *strategyIn*`=None)` → *FileObject*
Prepares a temporary work file for merge operations. The first argument can be either a string object or a file object. In the case of a string object, it is interpreted as the pathname to a file where the temporary merge file should be created. If it exists, it will be removed and re-created. In the case of a file object, the argument is assumed to be a previously opened file. Any write operations issued by TEXmerge will be executed against the passed file object.

`preambleCode`, if specified will be written at the beginning of the file in place of TEXmerge's normal preamble code. `formatIn` is currently unused. `strategyIn` determines the default form of preamble code to write. Valid values are `context`, `latex`, or `plain`.

`closeOutput(`*fileObject*`,` *postambleCode*`=None,` *formatIn*`=None,` *strategyIn*`=None,` *keepOpen*`=False)` → `None`
Completes preparation of a temporary work merge file for processing. *postambleCode* is written to the file if passed, otherwise an appropriate postamble will be supplied depending on the values of *formatIn* and *strategyIn*, if passed, or a default postamble will be written. The passed

fileObject will be closed unless *keepOpen* is passed as `True`.

`merge(`*targetPathname*`,` *mergeVariableDict*`,` *fileObject*`,` *options*`=0)` → `None`

Encapsulates the merge variables passed in *mergeVariableDict* for use in *targetPathname*. The merge variables are written to the merge work file as `\def` control sequences, and *targetPathname* is referenced via `\input` *targetPathname*.

Several merge options can be passed in the *options* argument:

1. `TXM_FRAMEVARS`: draw a box around every merged variable.
2. `TXM_DUPLEX`: assume the output will be printed on a duplexing device and insert `\eject` macros between merge invocations, when appropriate, to ensure that each merge invocation starts on the front side of the printed sheet.

`process(`*pathname*`,` *driverCommand*`,` *format*`,` *strategy*`)` → *IntegerObject*

Runs the TEX interpreter and a DVI backend against the merge work file *pathname*. The command used to run the TEX interpreter is derived from the *format* and *strategy* parameters. *Strategy* may be one of `context`, `latex`, or `plain`. If *strategy* is set to `context` then the environment variable `TEXENGINE` is used as the TEX processor, if set, or `texexec` otherwise. The DVI command string passed in *driverCommand* is used to run the DVI backend. It can contain a single `%s` which will be replaced with *pathname*. If no `%s` is present, *pathname* will be appended to *driverCommand*.

Returns the exit status of TEX interpreter or of the DVI backend command.

`processWithExtendedOutput(`*pathname*`,` *driverCommand*`,` *format*`,` *strategy*`)` → `(texstderr, texstdout, texlog, dvistderr, dvistdout)`

Works like the `process` function above, except for error handling. Failure of the TEX interpreter raises the exception `TeXException`. Failure of the DVI backend command raises the exception `DviException`. Successful completion of both the TEX interpreter and the DVI backend returns a tuple as above, providing complete diagnostics of the run.

`getNamedTextBlocks(`*pathname*`)` → {*block*$_1$: {*block*$_1$*AttrDict*}, ...}

Recursively scans *pathname* for occurrences of *named text blocks*, demarcated by the pair of macros `\StartNamedTextBlock[`*`attrName`*` = `*`value`*`, ...]` and `\EndNamedTextBlock`.

Text block attributes are as follows:

- `name` = Name of the text block,
- `seq` = *Integer*; several sections of text can be assigned the same name, but with unique sequence numbers. The extracted text will be a concatenation of like-named blocks, ordered by sequence number,
- *subkey*=*subvalue*; subkey name/value pairs provide a way to declare multiple blocks with the same name. Assigning differing name/value pairs makes each like-named block unique.

The class `TextBlockManager` can be used as an alternative to this function; it provides a simple frontend to this function's return value.

A.3 TEXmerge Class

The TEXmerge class provides an object-oriented interface to the module-level functions shown above.

Constructor

`TeXmerge(`*mergeTargetPathname*`=None,` *workPathname*`=None,` *mergeOptions*`=0,` *preambleCode*`=None,` *postambleCode*`=None,` *texmformat*`=None,` *`strategy`*`=None,` *keepIntermediateFiles*`=False)`

Methods

`setMergeTargetPathname(`*pathname*`)` → `None`

Sets the default merge target pathname for subsequent merge operations.

`setMergeOptions(`*self*`,` *mergeOptions*`)` → `None`

Sets the default merge options for future merge operations.

`setFormatAndStrategy(`*self*`,` *texmformat*`,` *strategy*`=None)` → `None`

Sets the default format and strategy to be used for future merge operations.

`probeMergeTargetAndSetFormat()` → `None`

Scans the current merge target pathname to determine the appropriate format and strategy that should be used during the `process()` method call.

`setFormatFromMergeTargetParentDirectory()` → `None`

Checks the merge target's parent directory for existence of the file `.texmformat`. If found, the contents of the file is assumed to be the format and strategy (specified similarly to the environment variable `TEXMFORMAT`) to be used when processing the merge file.

`getVariables()` → {*mergeVariableAttrDict*}

Calls the function `getExtendedNames` described above, passing the currently set merge target pathname as an argument. Returns the result of the call.

`openOutput(`*workPathnameOrFileObject*`=None)` → *FileObject*

Prepares the work file for subsequent merge operations. If no argument is passed, a default filename will be constructed.

`closeOutput()` → `None`

As above.

`merge(`*mergeVars*`=None,` *altMergeOptions*`=None,` *altMergeTargetPathname*`=None)` → `None`

Performs a merge operation using *mergeVars*, if passed, and alternate merge options and merge target pathname, also if passed.

`process(`*driverCommand*`)` → `None`

Run TEX interpreter according to currently set strategy and format. See `process` description above for details on the *driverCommand* string.

A.4 TextBlock Manager Class

Constructor

`TextBlockManager(`*pathname*`)`

Methods

`setPathname(`*pathname*`)` → `None`

Requests the `TextBlockManager` instance to scan *pathname* for named text blocks. Any information about previously scanned blocks is lost.

`getBlockNames()` → `[`$block_1$`,` $block_2$`, ...]`

Returns a list of the names of all named text blocks in the pathname last scanned.

`getBlock(`*blockName*`)` → *TextBlock*

Returns a `TextBlock` instance representation of the text block named *blockName*. Returns `None` if no such named block exists.

This same operation can be performed by using array indexing notation against the instance, i.e., indexing as with a dictionary object.

A.5 TextBlock Class

Constructor

`TextBlock(`*text-block-descriptor-dictionary*`)`

Methods

`getName()` → *StringObject*

Returns the instance's block name.

`getSubKeys()` → `[`$blockName_1$`, ...] | None`

Returns a list of unique subkey names associated with the given text blocks, or `None` if there are no associated subkeys.

`getSubkeyValues(`*subkeyName*`)` → `[`$subkeyName_1$`, ...]`

Returns a list of all the subkey values corresponding to the passed subkey name.

`getTextSegments(`*subkeyName*`=None,` *subkeyValue*`=None)` → {1: $textSeg_1$, 2: $textSeg_2$, ...}

Returns a dictionary of text segments, keyed by segment sequence number. The *subkeyName* and *subkeyValue* are optional; if specified, they are used to select the specific text block to access.

`getText(`*subkeyname*`=None,` *subkeyValue*`=None)` → *StringObject*

Returns a concatenation of all text segments in order by sequence number. The *subkeyName* and *subkeyValue* are optional; if specified, they are used to select the specific text block to access.

A.6 Exceptions

Exceptions can be raised by some of the class methods above. The exception objects have attributes which provide diagnostics about the associated error condition.

A.6.1 TeXException

This exception is raised when TEX cannot successfully interpret a file. Attributes:

- `stdout`: *StringObject* containing the standard output stream from the interpreter invocation,
- `stderr`: *StringObject* containing the standard error stream from the interpreter invocation,
- `logText`: *StringObject* containing the log file written by TEX.

A.6.2 DviException

This exception is raised when a DVI backend driver fails. Attributes:

- `stdout`: *StringObject* containing the standard output stream from the backend invocation,
- `stderr`: *StringObject* containing the standard error stream from the backend invocation.