BASIX — An Interpreter Written in \TeX

Andrew Marc Greene
MIT Project Athena, E40-342, 77 Massachusetts Avenue, Cambridge, MA 02139
617-253-7788. Internet: amgreene@mit.edu

Abstract
An interpreter for the BASIC language is developed entirely in \TeX. The interpreter presents techniques of scanning and parsing that are useful in many contexts where data not containing formatting directives are to be formatted by \TeX. \TeX’s expansion rules are exploited to provide a macro that reads in the rest of the input file as arguments and which never stops expanding.

Introduction
It is a basic tenet of the \TeX faith that \TeX is Turing-equivalent and that we can write any program in \TeX. It is also widely held that \TeX is “the most idiosyncratic language known to us.”

This project is an attempt to provide a simple programming front end to \TeX.

BASIC was selected because it is a widely used interpreted language. It also features an infix syntax not found in Lisp or PostScript. This makes it a more difficult but more general problem than either of these others.

The speed of the BASIX interpreter is not impressive. It is not meant to be. The purpose of this interpreter is not to serve as the BASIC implementation of choice. Its purpose is to display useful paradigms of input parsing and advanced \TeX programming.

Interaction with \TeX

Associative arrays. Using \csname it is possible to implement associative arrays in \TeX. Associative arrays are arrays whose index is not necessarily a number. As an example, if \student has the name of a student, we might look up the student’s grade with

\begin{verbatim}
\csname grade.\student\endcsname
\end{verbatim}

which would be \grade.Greene in my case. (In the case of \csname, all characters up to the \endcsname are used in the command sequence regardless of their category code.)

\begin{verbatim}
\end{verbatim}
Category codes. Normally, \TeX\ distinguishes between sixteen categories of characters. To avoid unwanted side effects, the \BASIX\ interpreter reassigns category codes of all non-letters to category 12, “other”. One undocumented feature of \TeX\ is that it will never let you strand yourself without an “active” character (which is normally the backslash); if you try to reassign the category of your only active character (with, e.g., \texttt{\catcode`\=12}), it will fail silently. To allow us to use every typeable character in \BASIX, we first make \texttt{\catcode31=0}, which then allows us to reassign the catcode of the backslash without stranding ourselves. (Of course, the \BASIX\ interpreter provides an escape back to \TeX\ which restores the normal category codes.)

We also change the category code of "\texttt{\-M}, the end-of-line character, to “active”. This lets us detect end-of-line errors that may crop up.

Semantics

Words. A \emph{word} is a collection of one or more characters that meet requirements based on the first character. The following table describes these rules using regular expression notation.\footnote{\emph{In this notation, }[A-Z, a-z] \emph{means “any character falling between A and Z or between a and z, inclusive.”}} These rules are:

<table>
<thead>
<tr>
<th>First Character Expression</th>
<th>Regular Expression</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>[A-Z, a-z]</td>
<td>[A-Z, a-z]</td>
<td>Identifier</td>
</tr>
<tr>
<td>[0-9]</td>
<td>[0-9]+</td>
<td>Integer</td>
</tr>
<tr>
<td>&quot;</td>
<td>&quot;[&quot;&quot;]*&quot;&quot;</td>
<td>String</td>
</tr>
<tr>
<td>Other</td>
<td></td>
<td>Symbol</td>
</tr>
</tbody>
</table>

An end-of-line at the end of a string literal is converted to a "\texttt{\textbackslash}.

Everything in \BASIX\ is one of these four types. Line numbers are integer literals, and both variables and commands are identifiers.

The \texttt{\scan} macro is used in \BASIX\ to read the next word. It uses \texttt{\futurelet} to look at the next token and determine whether it should be part of the current word; \emph{i.e.,} whether it matches the regular expression of the current type. If so, then it is read in and appended; otherwise \texttt{\scan} returns, leaving this next token in the input stream. The word is returned in the macro \texttt{\word}.

The peculiar way \texttt{\scan} operates gives rise to new problems, however. We can’t say

\begin{verbatim}
def\Goto{\scan[\line assignment]\word}
def\gopart{\after\scan[\line assignment]\word}
\end{verbatim}

because \texttt{\scan} looks at the tokens which follow it, which in this case are \texttt{\line assignment=\word}. We need some way to define the \texttt{\Goto} command so that the \texttt{\scan} is at the end of the macro; this will take the next tokens from the input stream. We therefore have \texttt{\scan “return”} to its caller by breaking the caller into two parts; the first part ends with \texttt{\scan} and the second part contains the code which should follow. The second macro is stored in \texttt{\afterscan}, and \texttt{\scan} ends with \texttt{\afterscan}. As syntactic sugar, \texttt{\after} has been defined as \texttt{\let\afterscan}. This allows \texttt{\Goto} to be coded as

\begin{verbatim}
def\Goto{\after\gopart\scan}
def\gopart{\line assignment=\word}
\end{verbatim}

(Actually, the \texttt{\Goto} code is slightly more complicated than this; but the scanner is the important point here.) This trick is used throughout the \BASIX\ interpreter to read in the next tokens from the user’s input without interrupting the expansion of the \TeX\ macros that comprise the interpreter.

Expressions. An \emph{expression} is a sequence of words that, roughly speaking, alternates between \emph{values} and \emph{operators}. Values fall into one of three categories: literals, identifiers, and parenthesized expressions. An operator is one of less-than, more-than, equality, addition, subtraction, multiplication, division, or reference. (Reference is an implicit operator that is inserted between a function identifier and its parameters.)

Expressions are evaluated in an approach similar to that used in the scanner. A word is scanned using \texttt{\scan} and its type is determined. “Left-hand” values are stored for relatives, additives, multiplicatives, and references. Using \TeX\’s grouping operations the evaluator is reentrant, permitting parenthesized expressions to be recursively evaluated.

In order to achieve a functionality similar to that of \texttt{\futurelet}, we exit the evaluator by \texttt{\expandafter\aftereval\word}, where the macro \texttt{\aftereval} is analogous to \texttt{\afterscan}. Since \texttt{\word} will contain neither macros nor tokens whose category codes need changing, this is as good as \texttt{\futurelet}.

---

2 In this notation, [A-Z, a-z] means “any character falling between A and Z or between a and z, inclusive.” An asterisk means “repeat the preceding specification as many times as needed, or never.” A plus means “repeat the preceding specification as many times as needed, at least once.” A question mark means “repeat the preceding specification zero or one times.” A dot means “any character.”
Structure of the Interpreter

The file basix.tex, which appears as an appendix to this paper, defines all the macros that are needed to run the interpreter. The last line of this file is \endeval, which is usually called when the interpreter has finished evaluating a line. In this case, it calls \enduserline, which, in turn, calls \parseline.

The \parseline macro is part of the Program Parser section of the interpreter. It starts by calling \scan to get the first word of the next line. If \word is an integer literal, it is treated as a line number and the rest of the input line is stored in the appropriate variable without interpretation. If \word is not an integer literal, it is treated as a command.

Each command is treated with an ad hoc routine near the bottom of basix.tex: however, most of them call on a set of utilities that appear earlier in the file.

Character-string calls. There is a simple library of macros that convert between ASCII codes and character tokens, test for string equality, take subsections of strings, and deal with concatenation.

Debugging definitions. The macro \div is a debugging-mode-only \immediate\write16 (hence the name \div). It is toggled by the user commands debug and nodebug.

Expression evaluation. The expression evaluator has a calling structure similar to that of the scanner. Calling routines are split in twain, with

\def\foo\{after\fooPartTwo\} \def\fooPartTwo\{etc\}

being a prototype of the calling convention. The evaluator will \scan as many words as it can that make sense; in contrast to the scanner, however, it evaluates each instead of merely accumulating them. This process is described above.

Linked list. The BASIX interpreter maintains a linked list of line numbers. The macro

\csname L(current line number)\endcsname

contains the next line number. These macros will follow the linked list (for the list command); they also can insert a new line or return the number of the next line.

Program parser. This section contains a number of critical routines. \eval is the macro that does the dispatching based on the user’s command. \mandatory specifies what the next character must be (for example, the character after the identifier in a let statement must be =). \parseline has already been described.

Syntactic scanner. This is the section containing \scan and its support macros, which are described above.

Type tests. These routines take an argument and determine whether it is an identifier, a string variable, a string literal, an integer literal, a macro, or a digit. The normal way of calling these routines is

\expandafter/\if\stringP\word

These predicates expand into either \tt (true) or \tf (false). Syntactic sugar is provided in the form of \istrue and \itsfalse. \ifstring\word doesn’t work because of the way \TeX matches \if and \fi tokens — only \if-style primitives are recognized.

User Utilities. This is the section of the interpreter in which most of the user commands are defined. Commands are preceded by \C (e.g., \Clist is the macro called when the user types list). Functions are preceded by \F.

Limitations of this implementation

BASIX is a minimal BASIC interpreter. There are enough pieces to show how things work, but not enough to do anything practical. Here is a description of the capabilities of this interpreter, so that the reader can play with it. Error recovery is virtually non-existent, so getting the syntax right and not calling non-existent functions is critical.

Entering programs. Lines beginning with an integer literal are stored verbatim. Lines are stored in ascending order, and if two or more lines are entered with the same number, only the last is retained.

Immediate commands. Lines not beginning with an integer are executed immediately. Colons are not supported, so only one command may appear on a line. (When a program line is executed, its line number is stripped and the remainder is executed as though it were an immediate command.)

Commands. The following commands are implemented in some form: goto, run, list, print, let, if, debug, nodebug, rem, system, exit, and stop (but not cont). The interpreter is case sensitive (although with an appropriate application of \uppercase it needn’t be; I was lazy), so these must be entered in lowercase.

The following tables list the commands with no parameters, the commands that take one parameter,
Andrew Marc Greene

and the two commands (let and if) that take special forms.

The commands with no parameters are:

- **run** Starts execution at the lowest line number.
- **stop** Stops execution immediately.
- **list** Lists all lines in order.
- **debug** Useful information sent to terminal.
- **nnodebug** Stop debugging mode.
- **rem** Rest of line is ignored.
- **system** Exit TiX.
- **exit** Exit BASIX to TiX.

The commands with one parameter are:

- **goto** Starts execution at the given line number.
- **list** Lists the given line.
- **print** Displays the given argument.

(Any of these arguments may be an expression.)

The **let** and **if** commands take special forms. Variable assignments require an explicit **let** command:

```plaintext
let (identifier) = (expression)
```

Conditionals do not have an **else** clause, and **goto** is not implied by **then**:

```plaintext
if (expression) then (new command)
```

The new command is treated as its own line.

**Expressions.** Expressions are defined explicitly above. The operators are +, -, *, /, <, =, and >. Parentheses may be used for grouping. Variables may not be referenced before being set. (Unlike in traditional BASIC, variables are not assumed to be 0 if never referenced, and they aren't cleared when **run** is encountered).

Functions are invoked with

```plaintext
(function name)((param),(param),...)
```

The parameters are implicitly-delimited expressions that are passed to \matheval (which is simply called \eval in the table below to save space). The following functions are defined:

- **len(string)** Returns number of characters in the string.
- **chr$(expr)** Returns the character with the given ascii value.
- **inc(expr)** Returns \eval(expr) + 1.
- **min(expr1,expr2)** Returns the lesser of two \evals.

**Generalization**

The BASIX interpreter can easily be generalized to serve other needs. These other needs might be to interpret Lisp or POSTSCRIPT code [Anyone want to write a POSTSCRIPT interpreter in TiX?]; or to take source code in a given language and pretty-print it.

The definition of a “word” can be changed by modifying the \scan macro. It selects a definition for \scantest based on the first character; \scantest is what determines if a given token matches the selected regular expression. The \scantest macro is allowed to redefine itself.

The evaluation of expressions can be extended or changed by modifying the \math code. Floating-point (or even fixed-point) numbers could be dealt with, although the period would need to pass the \digitP test in some cases and not in others.

The method of dealing with newlines is easily removed for languages such as POSTSCRIPT or Lisp for which all whitespace is the same.

**Obtaining copies of basix.tex**

The source code to this paper and the BASIX interpreter are available by anonymous ftp from gevalt.mit.edu, which is at IP address 18.72.1.4. I will also mail out copies to anyone without ftp abilities.

**Summary**

Using a number of TiX tricks, some more devious than others, a BASIC interpreter can be written in TiX. While TiX macros will often be less efficient than, for example, awk paired with TiX, solutions using only TiX will be more portable. A less general macro package than BASIX could be written that uses these routines as paradigms and that is very efficient at parsing a specific input format.
BASIX — An Interpreter Written in \TeX

Listing of basix.tex

1. ---basix.tex begins here.
2. %
3. %  BaSiX (with the emphasis on SICK!) by Andrew Marc Greene
4. %
5. %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
6. %
7. %  Andrew’s Affiliations
8. %
9. %  Copyright (C) 1990 by Andrew Marc Greene
10. %  <amgreene@mit.edu>
11. %  MIT Project Athena
12. %  Student Information Processing Board
13. %  All rights reserved.
14. %
15. %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
16. %
17. %  BaSiX’s Beginnings
18. %
19. \def \flageol {\catcode13=13}
20. \def \endflageol {\catcode13=5}
21. \def \struncat {\catcode'='#12}
22. \def \strcat {\catcode'='#11}
23. \flageol \let \eol \endflageol
24. \newif \ifresult
25. \newcount \xa \newcount \xb
26. \def \iwt {\immediate \uritel 6}
27. \def \empty {} 
28. \def \gobble #1 {} 
29. \def \spc {} 
30. \def \itstrue {tt} 
31. \def \itsfalse {tf} 
32. \def \isnull #1 {\resultfalse 
33. \expandafter \ifx \csname \string \empty #1 \endcsname \empty \resulttrue \fi} 
34. \newcount \matha \newcount \mathb
35. %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
36. %
37. %  Character-string Calls
38. %
39. %
40. \newcount \strtmp
41. \def \ascii #1 {\begin{group}{\ucode=65=#1\ucecase}{gdef \tmp{A}}{\endgroup} 
42. \def \strlen #1 {\strtmp -2 \don't count " \iw tokens 
43. \expandafter \if \string #1 \let \next \strIter \strIter #1 \iw \fi} 
44. \def \strIter #1 {\if x \iw #1 \let \next \relax \else \advance \strtmp by 1 \relax 
45. \fi \next} 
46. \def \Flen {\expandafter \strlen \expandafter {\Pa} \return {\number \strtmp}} 
47. \def \strcat 
48. \def \Fchar #1 {\expandafter \chr \expandafter {\Pa} \return {\tmp}} 
49. \def \struncat 
50. % first char only: 
51. % \def \Fasc {\expandafter \asc \expandafter {\Pa} \return {\number \strtmp}} 
52. %
53. %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
54. %
55. %  Debugging Definitions
56. %
57. %
58. \def \debug {\tracingmacros=2} 
59. \def \diw #1 {} 
60. \def \Cdebug {\let \diw \iw \tracingmacros=2 \endeval} 
61. \def \Cnodebug {\def \diw #1 {} \endeval}
62. \%
\def \expression {\let \afterexpression \afterscan \math)
\newcount \parens \newcount \mathParams
\def \math {\parens = 0 \mathParams 96 \mathInit \matheval)}
\def \mathRecurse {\advance \parens by 1 \relax \mathParams 96 \mathInit \matheval)}
\def \mathInit {\begingroup \let \mathAcc \empty \let \mathOpRel \empty \let \mathOpAdd \empty \let \mathOpMul \empty \let \mathlhFlef \empty)}
\def \matheval {\after \mathbranch \scan)}
\def \mathbranch {\diw {EXPRESS: \expandafter \noexpand \word:))
\let \next \matherr
\ifx \empty \uord \let \next \mathHardEnd \diw)}
\expandafter \if \numberP \uord \let \next \matiteral \diw)}
\expandafter \if \stringP \word \let \next \mathLiteral \fi
\expandafter \if \identifierP \word \let \next \mathIdentifier \fi
\expandafter \if \stringvarP \word \let \next \mathIdentifier \fi
\expandafter \if \macroP \word \let \next \mathMacro \fi
\ifx \word \0left \let \next \mathEndRecurse \fi
\ifx \word \0right \let \next \mathComma \fi
\if \word \Oplus \let \next \mathOp \diu{!+})
\ifx \word \Ominus \let \next \mathOp \diu{!-})
\ifx \word \Otimes \let \next \mathOp \diu{!*})
\ifx \uord \Odiv \let \next \mathOp \diw{!/})
\ifx \word \Olt \let \next \mathOp \diw{!<})
\ifx \word \Oeq \let \next \mathOp \did{!=})
\ifx \word \Ogt \let \next \mathOp \div(!>)
\if \word \Oleft \def \Oright {\def \Oleft {() \def \Oright {)) \def \Ocomma {,)}
\def \Oplus {+}\def \Ominus {-}\def \Otimes {*})\def \Odiv {/}
\def \Olt {<}\def \Oeq {=}\def \Ogt {>}
\def \Oleft {() \def \Oright {)}
\let \Oleft {() \def \Oright {)}
\let \Oleft {() \def \Oright {)}
\let \Oleft {() \def \Oright {)}
\let \Oleft {() \def \Oright {)}
\let \Oleft {() \def \Oright {)}
\let \Oleft {() \def \Oright {)}
\let \Oleft {() \def \Oright {)}
\let \Oleft {() \def \Oright {)}
BASIX – An Interpreter Written in TEX

\def\mathGT{\ifnum\matha>\mathb\matha-1\else\mathaO\fi)
\def\mathLT{\ifnum\matha<\mathb\matha-l\else\mathaO\fi)
\def\mathFlushRel{\mathFlushAdd\ifx\empty\mathOpRel\else
\matha=\mathlhRel\relax\mathb=\mathAcc\relax\math0pRel
\edef\mathAcc{\number\matha~\let\mathOpRel\empty\fi)
\def\mathFlushAdd{\mathFlushMul\ifx\empty\math0pAdd\else
\matha=\mathlhAdd\relax\mathb=\mathAcc\relax\math0pAdd
\edef\mathAcc{\number\matha\let\mathOpAdd\empty\fi)
\def\mathFlushMul{\mathFlushRef\ifx\empty\math~pMul\else
\matha=\mathlhMul\relax\mathb=\mathAcc\relax\mathOpMul
\edef\mathAcc~\number\matha~\let\mathOpMul\empty\fi)
\def\mathFlushRef{\ifx\empty\mathlhRef\else
\mathParam\mathlhRef \let\mathlhRef \empty\fi)
\def\mathop{\if \word+
\mathFlushAdd\let\mathlhAdd\mathAcc\let\mathOpAdd\mathAdd\fi
\if \word-
\mathFlushAdd\let\mathlhAdd\mathAcc\let\mathOpAdd\mathSub\fi
\if \word*
\mathFlushMul\let\mathlhMul\mathAcc\let\mathOpMul\mathMul\fi
\if \word/
\mathFlushMul\let\mathlhMul\mathAcc\let\mathOpMul\mathDiv\fi
\if \word=
\mathFlushRel\let\mathlhRel\mathAcc\let\mathOpRel\mathE\fi
\if \word>
\mathFlushRel\let\mathlhRel\mathAcc\let\mathOpRel\mathGT\fi
\if \word<
\mathFlushRel\let\mathlhRel\mathAcc\let\mathOpRel\mathLT\fi
\let\mathAcc\empty
\matheval)
\def \mathIdentifi er{%
\expandafter\ifx\csname C\word\endcsname\relax
\expandafter\ifx\csname F\word\endcsname\relax
\expandafter\ifx\csname V\word\endcsname\relax
\let\next\matherr\diw{LOSING:\word:)}
\else\let\next\mathVariable\fi
\else\let\next\mathFunction\fi
\else\let\next\mathCommand\fi
\next)
\def \mathIdentifier{%
\expandafter\ifx\csname surname\F\word\endcsname\relax
\expandafter\ifx\csname surname\V\word\endcsname\relax
\expandafter\ifx\csname surname\C\word\endcsname\relax
\let\next\matherr\diw{LOSING:\word:)
\else\let\next\mathVariable\fi
\else\let\next\mathFunction\fi
\else\let\next\mathCommand\fi
\next}
\def \mathCommand{\expandafter\edef\expandafter\word\expandafter\\
\{\csname V\word\endcsname\}\mathbranch}
\def \mathFunction{\expandafter\let\expandafter\math\HardEnd\word)}
\def \mathFunction{\expandafter\let\expandafter\math\Ref\mathlhRef)
\def \mathName{\expandafter\advance\expandafter\math\Params by 1\relax\chr\math\Params)
\def \PARAM{\tmp{\mathAcc:})}
\expandafter\expandafter\expandafter\expandafter\expandafter\tmp{\endcsname{\mathAcc})}
\def \mathComma{\expandafter\expandafter\expandafter\expandafter\expandafter\tmp{\endcsname{\mathAcc})}
\def \mathEndRecue{\mathEnd\advance\parens by -1\matheval)
\def \mathEnd{\div{\MATHEND: ACC=\mathAcc:)\mathFlushRel)
\def \mathtemp{\mathAcc\endgroup\def\mathAcc{\math\temp})
\def \math\HardEnd{\ifnum\parens>O\errmessage{Insuf-\relax
\let\next\endeval\else\let\next\mathFinal\fi\next)
\def \mathFinal{\mathEnd\let\value\mathAcc\endexpression)
\def \matherr{\errmessage{Syntax error: Unknown symbol \word))
\def \endexpression{\afterexpression)

TUGboat, Volume 11 (1990), No. 3 — Proceedings of the 1990 Annual Meeting
192. \%
193. \% Linked List
194. \%
195. \% \def\gotofirstline{\edef\lpointer{\csname LO\endcsname})
196. \% \def\foreachline#1{\ifnum\lpointer~99999\edef\word{\lpointer)#~%
197. \% \edef\lpointer{\csname L\word\endcsname\foreachline(#1)\fi)
198. \% \gotopast{#1} where #1 is a line number, will set \lpointer to
199. \% the least value such that L(\lpointer)>#1
200. \%
201. \% \def\gotopast#1{\def\lpointer{0)\def\target{#1)\gotopastloop)
202. \% \def \gotopastloop{\edef \tmp{\csname L\lpointer\endcsname)%
203. \% \ifnum\tmp<\target%
204. \% \edef\lpointer{\csname L\lpointer\endcsname)%
205. \% \let\next=\gotopastloop\else\let\next=\relax\fi
206. \% \flageol
207. \% \def \addLineToLinkedList#1#2
208. \% \{\def#i{#2)\diw{Just stored #2 in \noexpand #1)
209. \% now put it into linked list.
210. \% \expandafter\ifx\csname L\uord\endcsname\relax% if it isn't already there,
211. \% \gotopast{\word)\def\lpointer{what-should-point-to-word)
212. \% \expandafter\edef \csname L\word\endcsname{\csname L\lpointer\endcsname)
213. \% \expandafter\edef \csname L\lpointer\endcsname{\uord)
214. \% \f i
deval
215. \% \expandafter\def\csname LO\endcsname{99999)
216. \%
217. \%
218. \%
219. \%
220. \%
221. \%
222. \%
223. \%
224. \%
225. \%
226. \%
227. \%
228. \%
229. \%
230. \%
231. \%
232. \%
233. \%
234. \%
235. \%
236. \%
237. \%
238. \%
239. \%
240. \%
241. \%
242. \%
243. \%
244. \%
245. \%
246. \%
247. \%
248. \%
249. \%
250. \%
251. \% Syntactic Scanner
252. \%
253. \% The \scan routine reads the next WORD and then calls \afterscan.
254. \%
255. \% As syntactic sugar, one can write \after\foo to set \afterscan to
Here are the rules governing \texttt{WORD}. Initial whitespace is discarded. The word is the next single character, unless that character is one of the following:

- A-Z or a-z: \texttt{[A-Z,a-z][A-Z,a-z,0-9]*\$}
- 0-9: \texttt{[0-9]+}
- ":\texttt{[\"\"]+[\"\"]? (one or two; not the same if two)}

Note that the string literal ignores spaces but may be abnormally terminated by an end-of-line. (I wasn't sure how to express that as a regexp).

\newif\ifscan % shall we continue scanning?
\def\scan{}\futurelet\q\scanFirst}
\def\scanFirst{% Checks the first character to determine type.
\let\next\scanIter
\expandafter\if \spc\noexpand\q ~Space--ignore it
\let\next\scanSpace
\else
\if \eol\noexpand\q ~End of line--no word here
\let\next\scanEnd
\else
\ifcat A\noexpand\q % Then we have an identifier
\let\scanTest\scanIdentifier
\else
\expandafter\if \digit\q
\let\scanTest\scanNumericConstant
\else
\if It\noexpand\q
\let\scanTest\scanStringConstant
\else
\if \relation\noexpand\q
\let\scanTest\scanRelation
\else
\let\scanTest\scanfalse
\fi
\fi
\fi
\fi
\fi
\fi
\def\scanIdentifier{% Checks the first character to determine type.
\let\next\scanIter
\expandafter\if \spc\noexpand\q ~Space--ignore it
\let\next\scanSpace
\else
\if \eol\noexpand\q ~End of line--no word here
\let\next\scanEnd
\else
\ifcat A\noexpand\q % Then we have an identifier
\let\scanTest\scanIdentifier
\else
\expandafter\if \digit\q
\let\scanTest\scanNumericConstant
\else
\if It\noexpand\q
\let\scanTest\scanStringConstant
\else
\if \relation\noexpand\q
\let\scanTest\scanRelation
\else
\let\scanTest\scanfalse
\fi
\fi
\fi
\fi
\fi
\fi
\def\scanSpace{% If the first char is a space, gobble it and try again.
\def\scanIdentifier{\ifcat A\noexpand\q\scantrue\else
\expandafter\if \digit\q\scantrue
\else\if \noexpand\q\scantrue
\expandafter\def\expandafter\word\expandafter{\word #1}
\futurelet\q\scanContinueP}
\def\scanContinueP{\if \scanTest\ifscan\let\next\scanIter
\else\let\next\scanEnd\fi
\next}
\def\scanIdentifier{% Checks the first character to determine type.
\let\next\scanIter
\expandafter\if \spc\noexpand\q ~Space--ignore it
\let\next\scanSpace
\else
\if \eol\noexpand\q ~End of line--no word here
\let\next\scanEnd
\else
\ifcat A\noexpand\q % Then we have an identifier
\let\scanTest\scanIdentifier
\else
\expandafter\if \digit\q
\let\scanTest\scanNumericConstant
\else
\if It\noexpand\q
\let\scanTest\scanStringConstant
\else
\if \relation\noexpand\q
\let\scanTest\scanRelation
\else
\let\scanTest\scanfalse
\fi
\fi
\fi
\fi
\fi
\fi
\def\scanSpace{% If the first char is a space, gobble it and try again.
\def\scanIdentifier{\ifcat A\noexpand\q\scantrue\else
\expandafter\if \digit\q\scantrue
\else\if \noexpand\q\scantrue
\expandafter\def\expandafter\word\expandafter{\word #1}
\futurelet\q\scanContinueP}
\def\scanContinueP{\if \scanTest\ifscan\let\next\scanIter
\else\let\next\scanEnd\fi
\next}
\def\random{\relax\div{\texttt{SCANWW}:\word}}
\def\afterscan{\%}dumps trailing spaces.
\def\after{\%}
\def\after{\%}
\def\after{\%}
\def\after{\%}
\def\after{\%}
\def\relationP#1{\texttt{tf}} \% for now, only single-char relations
\def\identifierP#1{\texttt{\textbackslash expandafter\textbackslash identifierTest #1}}
\def\identifierTest#1\texttt{#2}\{\ifcat A#1\texttt{\textbackslash itstrue}\else\texttt{\textbackslash itsfalse}\fi\}
\def\stringvarP#1{\texttt{\textbackslash expandafter\textbackslash stringvarTest #1}}
\def\stringvarTest#1\texttt{#2}\{\if$#1\texttt{\textbackslash itstrue}\else\texttt{\textbackslash itsfalse}\fi\}
\def\stringP#1{\texttt{\textbackslash expandafter\textbackslash stringTest #1}}
\def\stringTest#1\texttt{#2}\{\if\texttt{\textbackslash itstrue}\else\texttt{\textbackslash itsfalse}\fi\}
\def\numberP#1{\texttt{\textbackslash expandafter\textbackslash numberTest #1}}
\def\numberTest#1\texttt{#2}\{\expandafter\if\texttt{\textbackslash digit #1}\texttt{\textbackslash itstrue}\else\texttt{\textbackslash itsfalse}\fi\}
\def\macroP#1{\texttt{\textbackslash expandafter\textbackslash macroTest #1}}
\def\macroTest#1\texttt{#2}\{\expandafter\if\texttt{\textbackslash relax}\texttt{\textbackslash itstrue}\else\texttt{\textbackslash itsfalse}\fi\}
\% 
\% \texttt{\textbackslash digit} tests its single-token argument and returns \texttt{tt} if true,
\% \texttt{tf} otherwise.
\% 
\% 
\% 
\% 
\% User Utilities. These are the commands that are called by the
\% user. We could really use a better section name. :-)
\% 
\% 
\% 
\% List (one line or all lines, for now)
\% 
\% 
\% Different degrees of \texttt{"stop execution"}
\% 
\% 
\% 
\% The command \texttt{"rem"} introduces a remark
\% 
\% 
\% 
\%
The ‘let’ command allows variable assignments

\def\letgetequals{\letgetvalue\mandatory{=}\after\letdoit\expression}
\def\letdoit{\edef\csname V\word\endcsname{\value}\after\endeval}

The ‘print’ command takes a list of expressions and displays them.

\def\Cprint{\after\printit\expression}
\def\printit{\iw{\value}\after\endeval}

Functions

Functions may read the counter \mathparams to find out the number of the top parameter. Parameters are in Pa Pb Pc etc.

\def\return{\expandafter\def\expandafter\mathAcc\expandafter)}
\def\Finc{\matha=\Pa\advance\matha by 1\return{\mnumber\matha}}
\def\Fmin{\ifnum\Pa<\Pb\return{\Pa}\else\return{\Pb}\fi}

Program execution control

\def\Crun{\let\endeval\endincrline\def\linen0{0}\after\evalline\scan\theline)
\def\Cgoto{\let\endeval\endgotoline\after\gotomain\scan}
\def\gotomain{\edef\lineno{\word}\after\endeval}
\def\execline{\message Executing line \lineno...}
\def\theline{\csname /\lineno\endcsname}
\message{THE LINE \theline}
\let\endeval\endincrline\after\evalline\expandafter\scan\theline)
\endfllageol

Different varieties of what to do at the end of a command:
Andrew Marc Greene

448. % get new line from user (enduserline)
449. % get next line in order (endincrline)
450. % get line in \lineno (endgotoline)
451. % keep parsing current line (endcolonline tbi)
452. %
453. \flageol%
454. \def\endincrline#1
455. {\diw{ENDINCRLINE}\edef\lineno{\csname L\lineno\endcsname}\execnextline}
456. %
457. \def\endgotoline#1
458. {\diw{ENDGOTOLINE}\let\endeval\execincrline\execnextline}%
459. %
460. \def\execnextline{\diw{Ready to execute line \lineno...} %
461. \ifnum\lineno<99999\let\next\execline\else\let\next\cleanstop\fi\next}%
462. %
463. \def\enduserline #1
464. {\diw{ENDUSERLINE}\parseline}\endflageol
465. %
466. \let\endeval\enduserline
467. %
468. %
469. % Start your engines!
470. %
471. \iw{This is BaSiX, v0.3, emphasis on the SICK! by amgreene@mit.edu}
472. \flageol
473. \catcode32=12
474. \endeval
475. \endeval
476. ---basix.tex ends here. The blank line at the end is significant.