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."[1] 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

```
\csname grade.\student\endcsname
```

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.)

BASIX makes extensive use of these arrays. Commands are begun with `C`; functions with `F`; variables with `V`; program lines with `/`; and the linked list of lines with `L`. This makes it easy for the interpreter to look up the value of any of these things, given the name as perceived by the user.

One benefit of `\csname...\endcsname` is that if the resultant command sequence is undefined, TEX replaces it with `\relax`. This allows us to check, using `\ifx`, whether the user has specified a non-existent identifier. This trick is used in exercise 7.7 in *The TEXbook*. We use it in BASIX to check for syntax errors and uninitialized variables.

Token streams. The BASIX interpreter was designed to be run interactively. It is called by typing `tex basix`; the file ends waiting for the first line of BASIC to be entered at TEX's * prompt. This also allows other files to `\input basix` and immediately follow it with BASIC code.

We cannot have the scanner read an entire line at once, since if the last line of `basix.tex` were a macro that reads a line as a parameter, we'd get a "`File ended while scanning use of \getline`" error. Instead, we use a method which at first blush seems more convoluted but which is actually simpler.

We note that TEX does not make any distinction between the tokens that make up our interpreter and the tokens that form the BASIC code. The BASIX interpreter is carefully constructed so that each macro ends by calling another macro (which may read parameters). Thus, expansion is never completed, but the interpreter can continue to absorb individual characters that follow it. These characters affect the direction of the expansion; it is this behaviour that allows us to implement a BASIC interpreter in TEX.

[1] Ward, Stephen A. and Robert H. Halstead, Jr., *Computation Structures*, MIT Press, 1990

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.*, `\catcode'\\=12`), it will fail silently. To allow us to use every typeable character in BASIX, we first make `\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 `^^M`, the end-of-line character, to "active". This lets us detect end-of-line errors that may crop up.

Semantics

Words. A *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.[2] These rules are:

First Character	Regular Expression	Meaning
`[A-Z,a-z]`	`[A-Z,a-z][A-Z,a-z,0-9]*$?`	Identifier
`[0-9]`	`[0-9]+`	Integer
`"`	`"[^"]*[",^^M]`	String
Other	`.`	Symbol

An end-of-line at the end of a string literal is converted to a ".

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

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

The peculiar way `\scan` operates gives rise to new problems, however. We can't say

```
\def\Cgoto{\scan\lineno=\word}
```

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

```
\def\Cgoto{\after\gotoPartTwo\scan}
\def\gotoPartTwo{\lineno=\word}
```

(Actually, the `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 *expression* is a sequence of words that, roughly speaking, alternates between *values* and *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 `\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 `\futurelet`, we exit the evaluator by `\expandafter\aftereval\word`, where the macro `\aftereval` is analogous to `\afterscan`. Since `\word` will contain neither macros nor tokens whose category codes need changing, this is as good as `\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 preceeding specification as many times as needed, at least once." A question mark means "repeat the preceeding 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 `\diw` is a debugging-mode-only `\immediate\write16` (hence the name `\diw`). 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\expression}
\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. `\evalline` 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 `\itstrue` 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,

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. Does *not* clear variables.
`stop`	Stops execution immediately.
`list`	Lists all lines in order.
`debug`	Useful information sent to terminal.
`nodebug`	Stop debugging mode.
`rem`	Rest of line is ignored.
`system`	Exit TEX.
`exit`	Exit BASIX to TEX.

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:

`let` ⟨*identifier*⟩ `=` ⟨*expression*⟩

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

`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

⟨*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 `\eval`s.

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 TEX?]; 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 TEX tricks, some more devious than others, a BASIC interpreter can be written in TEX. While TEX macros will often be less efficient than, for example, `awk` paired with TEX, solutions using only TEX 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.

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
24. \endflageol
25. \newif\ifresult
26. %\newcount\xa\newcount\xb
27. \def\iw{\immediate\write16}
28. \def\empty{}
29. \def\gobble#1{}
30. \def\spc{ }
31. \def\itstrue{tt}
32. \def\itsfalse{tf}
33. \def\isnull#1{\resultfalse
34. \expandafter\ifx\csname empty#1\endcsname\empty\resulttrue\fi}
35. \newcount\matha\newcount\mathb
36. %
37. %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
38. %
39. %  Character-string Calls
40. %
41. \newcount\strtmp
42. \def\ascii#1{\strtmp`#1}
43. \def\chr#1{\begingroup\uccode65=#1\uppercase{\gdef\tmp{A}}\endgroup}
44. \def\strlen#1{\strtmp-2% don't count " " \iw tokens
45.   \expandafter\if\stringP #1\let\next\strIter\strIter #1\iw\fi}
46. \def\strIter#1{\ifx\iw#1\let\next\relax\else\advance\strtmp by 1\relax
47.   \fi\next}
48. \def\Flen{\expandafter\strlen\expandafter{\Pa}\return{\number\strtmp}}
49. \strcat
50. \def\F$chr${\expandafter\chr\expandafter{\Pa}\return{\tmp}}
51. \struncat
52. % first char only:
53. % \def\Fasc{\expandafter\asc\expandafter{\Pa}\return{\number\strtmp}}
54. %
55. %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
56. %
57. %  Debugging Definitions
58. %
59. \def\debug{\tracingmacros=2}
60. \def\diw#1{}
61. \def\Cdebug{\let\diw\iw\tracingmacros=2 \endeval}
62. \def\Cnodebug{\def\diw##1{}\endeval}
63. %
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%  Expression Evaluation
%
\def\expression{\let\afterexpression\afterscan\math}
%
%  (math is a misnomer and should -> expr)
%
\newcount\parens\newcount\mathParams
\def\math{\parens=0\mathParams96\mathInit\matheval}
\def\mathRecurse{\advance\parens by 1\relax\mathParams96\mathInit\matheval}
\def\mathInit{\begingroup
  \let\mathAcc\empty
  \let\mathOpRel\empty
  \let\mathOpAdd\empty
  \let\mathOpMul\empty
  \let\mathlhRef\empty}
%
\def\matheval{\after\mathbranch\scan}
%
\def\mathbranch{\diw{EXPRESS:\expandafter\noexpand\word:}
\let\next\matherr
\ifx\empty\word\let\next\mathHardEnd\else                  % Expr. end?
\expandafter\if\numberP\word\let\next\mathLiteral\diw{@@}\fi  % Number?
\expandafter\if\stringP\word\let\next\mathLiteral\fi       % String literal?
\expandafter\if\identifierP\word\let\next\mathIdentifier\fi % Identifier?
\expandafter\if\stringvarP\word\let\next\mathIdentifier\fi  % :-(
\expandafter\if\macroP\word\let\next\mathMacro\fi          % Macro?
\ifx\word\Oleft\let\next\mathRecurse\fi                    % Open paren?
\ifx\word\Oright\let\next\mathEndRecurse\fi                % Close paren?
\ifx\word\Ocomma\let\next\mathComma\fi                     % Comma?
%
%  Operator?
%
\ifx\word\Oplus\let\next\mathOp\diw{!+}\fi
\ifx\word\Ominus\let\next\mathOp\diw{!-}\fi
\ifx\word\Otimes\let\next\mathOp\diw{!*}\fi
\ifx\word\Odiv\let\next\mathOp\diw{!/}\fi
\ifx\word\Olt\let\next\mathOp\diw{!<}\fi
\ifx\word\Oeq\let\next\mathOp\diw{!=}\fi
\ifx\word\Ogt\let\next\mathOp\diw{!>}\fi
%
\fi\next}
%
\def\Oleft{(}\def\Oright{)}\def\Ocomma{,}
\def\Oplus{+}\def\Ominus{-}\def\Otimes{*}\def\Odiv{/}
\def\Olt{<}\def\Oeq{=}\def\Ogt{>}
%
% There's got to be a better way to do the above....
%
\def\mathLiteral{\diw{MLIT}\ifx\empty\mathAcc\diw{ACCUMUL:\word:}
\expandafter\def\expandafter\mathAcc\expandafter
{\expandafter\expandafter\expandafter\empty\word}
\else
\diw{ACC has :\mathAcc: and word is :\word:}
\errmessage{Syntax Error: Two values with no operator}\fi\matheval}
%
%  Operator stuff:  (Need to add string support / error checking)
%
\def\mathAdd{\advance\matha by \mathb}
\def\mathSub{\advance\matha by -\mathb}
\def\mathMul{\multiply\matha by \mathb}
\def\mathDiv{\divide\matha by \mathb}
\def\mathEQ{\ifnum\matha=\mathb\matha-1\else\matha0\fi}
```

```
\def\mathGT{\ifnum\matha>\mathb\matha-1\else\matha0\fi}
\def\mathLT{\ifnum\matha<\mathb\matha-1\else\matha0\fi}
%
\def\mathFlushRel{\mathFlushAdd\ifx\empty\mathOpRel\else
  \matha=\mathlhRel\relax\mathb=\mathAcc\relax\mathOpRel
  \edef\mathAcc{\number\matha}\let\mathOpRel\empty\fi}
%
\def\mathFlushAdd{\mathFlushMul\ifx\empty\mathOpAdd\else
  \matha=\mathlhAdd\relax\mathb=\mathAcc\relax\mathOpAdd
  \edef\mathAcc{\number\matha}\let\mathOpAdd\empty\fi}
%
\def\mathFlushMul{\mathFlushRef\ifx\empty\mathOpMul\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\mathEQ\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\mathIdentifier{%
\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\mathVariable{\expandafter\edef\expandafter\word\expandafter
{\csname V\word\endcsname}\mathbranch}
\def\mathCommand{\expandafter\mathHardEnd\word}
\def\mathFunction{\expandafter\let\expandafter\mathlhRef
\csname F\word\endcsname\matheval}
%
\def\mathParam{\advance\mathParams by 1\relax\chr\mathParams
  \diw{PARAM:\tmp:\mathAcc:}
  \expandafter\edef\csname P\tmp\endcsname{\mathAcc}}
\def\mathComma{\mathEnd\mathParam\mathInit\matheval}
\def\mathEndRecurse{\mathEnd\advance\parens by -1\matheval}
\def\mathEnd{\diw{MATHEND: ACC=\mathAcc:}\mathFlushRel
  \xdef\mathtemp{\mathAcc}\endgroup\edef\mathAcc{\mathtemp}}
\def\mathHardEnd{\ifnum\parens>0\errmessage{Insufficient closeparens.}\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}
```

```
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%  Linked List
%
\def\gotofirstline{\edef\lpointer{\csname L0\endcsname}}
\def\foreachline#1{\ifnum\lpointer<99999\edef\word{\lpointer}#1%
\edef\lpointer{\csname L\word\endcsname}\foreachline{#1}\fi}
%
%  gotopast{#1}  where #1 is a line number, will set \lpointer to
%                the least value such that L(lpointer)>#1
%
\def\gotopast#1{\def\lpointer{0}\def\target{#1}\gotopastloop}
%
\def\gotopastloop{\edef\tmp{\csname L\lpointer\endcsname}%
\ifnum\tmp<\target%
\edef\lpointer{\csname L\lpointer\endcsname}%
\let\next=\gotopastloop\else\let\next=\relax\fi
\next}
%
\flageol\def\addLineToLinkedList#1#2
{\def#1{#2}\diw{Just stored #2 in \noexpand #1}%
% now put it into linked list...
\expandafter\ifx\csname L\word\endcsname\relax% if it isn't already there,
\gotopast{\word}% \def\lpointer{what-should-point-to-word}
\expandafter\edef\csname L\word\endcsname{\csname L\lpointer\endcsname}%
\expandafter\edef\csname L\lpointer\endcsname{\word}%
\fi\endeval
}\endflageol
\expandafter\def\csname L0\endcsname{99999}
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%  Program Parser
%
\def\evalline{%\iw{EVALLINE :\word:}%
\csname C\word\endcsname}  %error-checking?  :-)
\def\evalerror{\errmessage{Unkonwn command.  Sorry.}}
%
%  \mandatory takes one argument and checks to see if the next
%  non-whitespace token matches it.  If not, an error is generated.
%
\def\mandatory#1{\def\tmp{#1}\mandatest}
\def\mandatest#1{\def\tmpp{#1}\ifx\tmp\tmpp\let\next\afterscan\else
\let\next\manderror\fi\next}
\def\manderror{\errmessage{\tmpp\spc read when \tmp\spc expected.}%
\afterscan}
%
%  \parseline gets the first WORD of the next line.  If it's a line
%  number, \scanandstoreline is called; otherwise the line is executed.
%
\def\parseline{\after\firsttest\scan}
\def\firsttest{\expandafter\if\numberP\word
\let\next\grabandstoreline\else\let\next\evalline\fi\next}
\def\grabandstoreline{\diw{Grabbing line \word.}%
\expandafter\addLineToLinkedList\csname/\word\endcsname}
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%  Syntactic Scanner
%
%  The \scan routine reads the next WORD and then calls \afterscan.
%
%  As syntactic sugar, one can write \after\foo to set \afterscan to
```

```
% \foo.
%
% Here are the rules governing 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:  [A-Z,a-z][A-Z,a-z,0-9]*\$?
%  0-9:         [0-9]+
%  ":           "[^"]*"
%  <,=,>:       [<=>][<=>]?  (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{\def\word{}\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"\noexpand\q
  \let\scanTest\scanStringConstant\else
\expandafter\if\relationP\q
  \let\scanTest\scanRelation
\else
  \let\scanTest\scanfalse
\fi\fi\fi\fi\fi\fi\next}
%
\def\scanIter#1{\expandafter\def\expandafter\word\expandafter{\word #1}
  \futurelet\q\scanContinueP}
\def\scanContinueP{\scanTest\ifscan\let\next\scanIter
  \else\let\next\scanEnd\fi\next}
%
\def\scanSpace#1{\scan}% 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$\noexpand\q\scantrue
  \expandafter\def\expandafter\word\expandafter{\expandafter$\word}
  \let\scanTest\scanfalse\else
\scanfalse\fi\fi\fi}
\def\scanEndString{\scanfalse}
\def\scanNumericConstant{\expandafter\if\digit\q\scantrue\else\scanfalse\fi}
\def\scanStringConstant{\scantrue\if"\q\let\scanTest\scanfalse\fi}
\def\scanRelation{\if<\q\scantrue\else\if>\q\scantrue\else\if=\q\scantrue
\else\scanfalse\fi\fi\fi}
%
\def\scanEnd#1{\relax\diw{SCANNED:\word:}
\afterscan #1}% dumps trailing spaces.
\def\after{\let\afterscan}%
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%  Type Tests (Predicates for type determination)
%
```

```
\def\relationP#1{tf}  % for now, only single-char relations
\def\identifierP#1{\expandafter\identifierTest #1\\}
\def\identifierTest#1#2\\{\ifcat A#1\itstrue\else\itsfalse\fi}
\def\stringvarP#1{\expandafter\stringvarTest #1\\}
\def\stringvarTest#1#2\\{\if$#1\itstrue\else\itsfalse\fi}
\def\stringP#1{\expandafter\stringTest #1\\}
\def\stringTest#1#2\\{\if #1"\itstrue\else\itsfalse\fi}
\def\numberP#1{\expandafter\numberTest #1\\}
\def\numberTest#1#2\\{\expandafter\if\digit #1\itstrue\else\itsfalse\fi}
\def\macroP#1{\expandafter\macroTest #1\\}
\def\macroTest#1#2\\{\expandafter\ifx #1\relax\itstrue\else\itsfalse\fi}
%
%  \digit tests its single-token argument and returns tt if true,
%  tf otherwise.
%
%
\def\digit#1{%
\if0\noexpand#1\itstrue\else
\if1\noexpand#1\itstrue\else
\if2\noexpand#1\itstrue\else
\if3\noexpand#1\itstrue\else
\if4\noexpand#1\itstrue\else
\if5\noexpand#1\itstrue\else
\if6\noexpand#1\itstrue\else
\if7\noexpand#1\itstrue\else
\if8\noexpand#1\itstrue\else
\if9\noexpand#1\itstrue\else\itsfalse
\fi\fi\fi\fi\fi\fi\fi\fi\fi\fi
% 9  8  7  6  5  4  3  2  1  0
}
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%  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)
%
\def\Clist{\after\listmain\scan}
\def\listmain{\isnull{\word}\ifresult\let\next\listalllines
\else\let\next\listoneline\fi\next}
\def\listline{\iw{\word\spc\csname/\word\endcsname}}
\def\listalllines{\gotofirstline\foreachline{\listline}\endeval}
\def\listoneline{\listline\endeval}
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%  Different degrees of ``stop execution''
%
\def\Csystem{\end}       % exits to the system
\def\Cexit{}% \endflageol} % exits to TeX
\flageol%
\def\Cstop#1
{\iw{Stopped in \lineno.}\cleanstop}%
\def\cleanstop{\diw{CLEANSTOP}\let\endeval\enduserline\endeval
}\endflageol
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%  The command ``rem'' introduces a remark
%
\def\Crem{\endeval}%
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
%
%  The ''let'' command allows variable assignments
%
\def\Clet{\after\letgetequals\scan}
\def\letgetequals{\after\letgetvalue\mandatory{=}}
\def\letgetvalue{\after\letdoit\expression}
\def\letdoit{\expandafter\edef\csname V\word\endcsname{\value}%
\endeval}
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%  The ''print'' command takes a [list of] expression[s] and displays
%  it [them].
%
\def\Cprint{\after\printit\expression}
\def\printit{\iw{\value}\endeval}
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%  The ''if'' command takes an expression, the word ''then,'' and
%  another command.  If the expression is non-zero, the command is
%  executed; otherwise it is ignored.
%
\def\Cif{\after\getift\expression}
\def\Cthen{\errmessage{Syntax error: THEN without IF}}
\def\getift{\after\getifh\mandatory t}
\def\getifh{\after\getife\mandatory h}
\def\getife{\after\getifn\mandatory e}
\def\getifn{\after\consequent\mandatory n}
\def\consequent{\ifnum\value=0\let\next=\endeval\else\let\next=\evalconsq\fi
\next}
\def\evalconsq{\after\evalline\scan}
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%  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\lineno{0}\endeval}
\def\Cgoto{\let\endeval\endgotoline\after\gotomain\scan}
\def\gotomain{\edef\lineno{\word}\endeval}
%
\flageol%
\def\execline{%\message{Executing line \lineno...}%
\edef\theline{\csname/\lineno\endcsname}%
%\message{THE LINE\theline}%
\let\endeval\endincrline\after\evalline\expandafter\scan\theline
}\endflageol
%
%  Different varieties of what to do at the end of a command:
%
```

```
% get new line from user  (enduserline)
% get next line in order  (endincrline)
% get line in \lineno     (endgotoline)
% keep parsing current line (endcolonline tbi)
%
\flageol%
\def\endincrline#1
{\diw{ENDINCRLINE}\edef\lineno{\csname L\lineno\endcsname}\execnextline}
%
\def\endgotoline#1
{\diw{ENDGOTOLINE}\let\endeval\execincrline\execnextline}%
%
\def\execnextline{\diw{Ready to execute line \lineno...}%
\ifnum\lineno<99999\let\next\execline\else\let\next\cleanstop\fi\next}%
%
\def\enduserline #1
{\diw{ENDUSERLINE}\parseline}\endflageol
%
\let\endeval\enduserline
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%   Start your engines!
%
\iw{This is BaSiX, v0.3, emphasis on the SICK!  by amgreene@mit.edu}
\flageol
\catcode32=12
\endeval

---basix.tex ends here.  The blank line at the end is significant.
```