LIM and Nanoweb

Greg Nelson
Imaging Systems Laboratory
HP Laboratories Palo Alto
HPL-2005-41
February 28, 2005*

LIM, Nanoweb, Literate Programming, Law of the Excluded Miracle

This paper has several aims: It introduces two useful tools to the programming community, it argues against the "Law of the Excluded Miracle", and it illustrates a lightweight style of literate programming that the author has found effective and useful.

LIM (the Language of the Included Miracle) is a novel little programming language useful for writing small programs that include a good measure of syntactic processing. A LIM program is like a context free grammar, but with deterministic choice instead of non-deterministic choice and with both input and output as opposed to grammers, which support input only (or output only if they are viewed as generators). An alternative one sentence description: start with Dijkstra's calculus of guarded commands, drop the Law of the Excluded Miracle, and allow the commands of the calculus to operate on a state space containing two character streams, an input stream and an output stream.

Nanoweb is an ultralight literate programming tool implemented in LIM. Nanoweb is useful, and also serves in this paper in the role of two examples: an example of a LIM program and as an example of a lightweight literate program.

* Internal Accession Date Only
Approved for External Publication

LIM and Nanoweb

Greg Nelson

February 28, 2005

Abstract. This paper has several aims: It introduces two useful tools to the programming community, it argues against the "Law of the Excluded Miracle", and it illustrates a lightweight style of literate programming that the author has found effective and useful.

LIM (the Language of the Included Miracle) is a novel little programming language useful for writing small programs that include a good measure of syntactic processing. A LIM program is like a context free grammar, but with deterministic choice instead of non-deterministic choice and with both input and output as opposed to grammers, which support input only (or output only if they are viewed as generators). An alternative one sentence description: start with Dijkstra's calculus of guarded commands, drop the Law of the Excluded Miracle, and allow the commands of the calculus to operate on a state space containing two character streams, an input stream and an output stream.

Nanoweb is an ultralight literate programming tool implemented in LIM. Nanoweb is useful, and also serves in this paper in the role of two examples: an example of a LIM program and as an example of a lightweight literate program.

1 Introduction

To meet the enormous challenge presented by software technology, we scientists and engineers will need to embrace some radical new ideas. One idea that goes to the root of the problem is Donald E. Knuth's *Literate programming* [8, 10], the idea that a program should be readable as a literary work. Knuth's system Web produces, from a single source file, either an executable or a typeset exposition of a program. His first version of Web targetted Pascal (that is, used a Pascal compiler in the path to the executable and used an augmented version of the Pascal language in the exposition); subsequent versions of Web, and subsequent literate programming tools by others who have followed Knuth's lead, have targetted other programming languages. Since Knuth's first work, the trend in literate programming tools has been towards increasingly lightweight systems; for example, consider Norman Ramsey's recent system `noweb` [14].

This paper describes Nanoweb, an ultralightweight literate programming tool. For simplicity, Nanoweb omits the reordering and the macro facilities supported by most other literate programming tools (even `noweb`). Nanoweb was written by the author in

2003 to meet the immediate needs of the research project Denali-2 [5] (since discontinued), and my complexity budget was extremely tight (I budgeted 500 lines for code, exposition, documentation, and tests).

As released, Nanoweb targets Java, since that was to be the implementation language of Denali-2, but Nanoweb is almost language-independent, and this paper will explain how to retarget it to other languages. Nanoweb is implemented in LIM, the Language of the Included Miracle designed and implemented in 1993 by Mike Burrows and Greg Nelson, and available for download over the internet [2]. This paper describes LIM and Nanoweb, using the LIM implementation of Nanoweb as an example. The opening paragraphs of this introduction have focussed on Nanoweb, but many readers may find the description of LIM to be the most interesting part of the paper. The paper consists of the Nanoweb and LIM specifications (also known as man pages), followed by the LIM implementation of Nanoweb as typeset by itself. Thus this paper is about two parts man page and one part lightweight literate program. Before the current literate programming movement, the Unix man page had already emerged as a useful and distinctive genre (and a successful use of specifications in software engineering), a fact that seems important and under-appreciated, and I would like in this paper to give that genre its due. The paper is stitched together with twenty-six additional sentences of commentary, of which sixteen have formed this introduction.

2 The Nanoweb manpage

2.1 Name

`nanoweb` – an ultralightweight literate programming tool.

2.2 Syntax.

```
nanoweb < rawprogfile > texfile.
```

2.3 Explanation.

Nanoweb translates the raw source of a Java program into LaTeX commands that will typeset the program for a discerning reader. Nanoweb will also work with programming languages that share Java's syntax for comments and for string literals, such as C and C++. The program author must write and typeset the exposition, which she includes in the comments in the raw source.

More precisely, the exposition should be placed in *exposition comments*, which are defined as comments opened by `/*` whose openiing delimiters have nothing but white space preceding them on their lines. Nanoweb copies the contents of exposition comments in *comment mode*, which means it copies them with almost no transformation. In comment mode, Nanoweb will copy any LaTeX commands from input to output, so that, if you want to, you can use all the power of LaTeX (sectioning commands, math mode, bulleted paragraphs, bells, whistles) to typeset the exposition. In particular, you can make tables or use LaTeX verbatim mode. Therefore, Nanoweb has no special

support for verbatim displays within comment mode (unlike its predecessor `m3totex` and its cousins `ppmp` and `pplim`).

Input text that isn't in exposition comments (sometimes called *code*) is translated in *code mode*, which means that it is typset in typewriter font, with spacing, line breaks, and indentation preserved. In code mode, Nanoweb will carefully quote any characters that are special to TeX, so you can't do your own typesetting in code mode: the reader sees what the compiler sees. A comment that isn't an exposition comment will also be translated in code mode. In this case, Nanoweb requires that the comment begin and end on the same source line.

Nanoweb doesn't introduce or remove newlines, so LaTeX error message line numbers given relative to Nanoweb's output are also accurate relative to Nanoweb's input.

Nanoweb gives special treatment only to comments delimited by `/*` and `*/`; comments delimited by `//` and end-of-line are ignored.

Nanoweb supports three additional features:

2.4 Preamble.

The portion of the raw program source up to and including the first group of one or more blank lines is called the *preamble*. (A blank line is a line containing spaces and tabs only.) `nanoweb` reads and ignores the preamble and starts producing TeX commands from the position of the raw source immediately after the preamble. This allows you to include at the beginning of your raw source things like author information, copyright notices, change logs, and the like. Just put those things at the top of your file, and make sure they have a blank line after them (and no internal blank lines).

2.5 LaTeX boilerplate.

By default, Nanoweb will produce LaTeX commands to begin and end the document, and to select the LaTeX "article" style. This may be undesirable, for example, for users of other TeX macro packages, or if Nanoweb's output is to be included in a longer document (as it is in this paper). Therefore, Nanoweb will omit the LaTeX boilerplate at the beginning and ending of its output if the one-word comment "`noboilerplate`" (with no spaces in or around it) appears immediately after the preamble.

2.6 Escape to code mode.

When the exposition refers to a program name, it is a good idea to typeset the name in typewriter font rather than roman font, to help the reader recognize that the name is a name. This leads to many instances of the pattern '`\hbox{\tt ...}`' in the exposition, which are tedious to type and distracting to read. (Of course, we never read the raw source, but we never say never, either.) To alleviate this problem, Nanoweb has the following feature: in comment mode, it will put any phrase enclosed in double-quotes into a TeX box in typewriter font. So instead of

```
\hbox{\tt p.link}
```

you can just write

```
"p.link"
```

(including the double-quotes). Any TeX special characters in the double-quoted phrase are quoted, just as in code mode. The double-quoted phrase must appear entirely on a single line of the raw source. If for some reason you want a literal double-quote to appear in your exposition, you will need to use a construction like "`\char'042`" (double-quote is character octal forty-two). If you want to employ the double-quote character in some non-trivial way in your own typesetting in comment mode (for example, to use TeX's notation for hexadecimal numbers), you will just have to find a way to do without it (for example, by using octal notation instead).

2.7 Author

Greg Nelson designed and implemented Nanoweb, and wrote this man page.

3 The LIM manpage

3.1 Name.

lim – text processing language with pattern matching and backtracking

3.2 Syntax.

```
lim progfile
```

3.3 Description.

Parse and execute the LIM program in the file `progfile`. The LIM program will read from standard input and write to standard output.

3.4 LIM language overview

LIM is a text processing language in the general family of YACC, LEX, sed, and AWK, but is based on Dijkstra's calculus of guarded commands. The name is an acronym for Language of the Included Miracle, because LIM programs can violate the so-called Law of the Excluded Miracle. This law was introduced in Edsger W. Dijkstra's classic *Discipline of Programming* [3]. The generalized calculus without the law was introduced in a paper of Greg Nelson's [12]. Another of Nelson's papers provides an introduction and overview of the generalized calculus [13].

The input to YACC and other conventional parser generators is a hybrid of two rather different languages: a BNF-like language to specify syntax and a conventional imperative programming language to specify actions to be performed during parsing. The LIM approach is different: it provides a single language that is almost as convenient as BNF at describing language syntax and almost as convenient as conventional

imperative programming languages at describing actions to be performed during parsing. This generality is obtained not by adding features but by dropping a restriction, namely the Law of the Excluded Miracle.

LIM is a minimalist language and there are many situations where it fails to be applicable because of it lacks some crucial feature. But when it is applicable, it can produce solutions of attractive simplicity. After completing this overview of LIM, the man page provides a complete description.

Here are four important LIM primitives:

`Wr(s)`	Write `s`
`Rd(s)`	Read `s`, failing if it is not next in the input
`A ; B`	Do `A`, then do `B`
`A | B`	Do `A`, else do `B` if `A` fails.

In the first two commands, `s` is a string literal (for now). The command `Rd(s)` reads `s` from the input if it is present, otherwise it fails. Failure does not cause an error, it causes the implementation to backtrack and find another path that avoids failure. The command `Wr(s)` writes the string `s` to the output; it never fails. The command `A;B` fails if either `A` or `B` fail, while the command `A|B` fails only if both `A` and `B` fail. For example, the command

```
Rd("0"); Wr("0") | Rd("1"); Wr("1")
```

will copy a single binary digit from input to output, or fail if the input is exhausted or the next input character is not a binary digit. (The semicolon binds more tightly than the vertical bar.)

The Law of the Excluded Miracle holds that wp(A, **false**) is **false** for any command A, where in general wp(A, R) holds of those input states from which all computational outcomes of A are output states that satisfy R. Now it is clear that if there can be an input state that is related to no outcomes by A, then such a state will satisfy wp(A, R) for any R, even for $R :=$ **false**, since the quantification over all its outcomes is vacuous. (No state satisfies **false**, but every element of the empty set satisfies **false**.) In short: dropping the Law of the Excluded Miracle is equivalent to enriching the calculus with commands that correspond to partial relations between input states and outcomes. This generalization is appropriate for programming parsers. From an input state in which the input is exhausted or the string `s` is not next in the input, we would like to say that it is not possible to execute the command `Rd(s)`: the command relates such an input state to no outcomes at all. If one insists on an operational interpretation, the best we can say is that the effect is to backtrack, but the semantic model with partial relations is clearer than the operational semantics.

Failing commands never have side effects. If a computation has side effects and then fails, it is as though the side effects never happened. For example,

```
Rd("0"); Wr("0")
```

and

```
Wr("0"); Rd("0")
```

mean exactly the same thing: copy a zero from input to output if possible, and otherwise fail without writing anything.

The command `A | B` is pronounced "A else B". The operation is called *deterministic choice*. It is generally more useful than non-deterministic choice, so it is ironic that non-deterministic choice is so much better known. For example, non-deterministic choice is used in regular expressions, in Dijkstra's classic calculus of guarded commands, and in the conventional BNF notation for context-free grammars. The reason may be that deterministic choice is interesting only in the absence of the Law of the Excluded Miracle (if `A` obeys the Law, then `A` never fails, and `A | B` is equivalent to `A`).

Here are four more LIM primitives, three of which are overloadings of of the previous `Rd` and `Wr` primitives:

`Wr(c)`	write the character `c`
`Rd(c)`	read the character `c`
`Rd()`	read and return the next input character
`DO A OD`	repeatedly execute `A` until it fails

`Rd()` fails if the input is at end of file; otherwise it returns the (integer code for) the next character in the input, as it consumes that character. `Rd(c)` reads the character `c` from the input or fails if the next character isn't `c`. `Wr(c)` never fails; it writes to output the character `c`.

For example, the following program will copy input to output, replacing all occurrences of "`gnelson`" with "`burrows`" and "`burrows`" with "`gnelson`":

```
DO Rd("burrows"); Wr("gnelson")
 | Rd("gnelson"); Wr("burrows")
 | Wr(Rd())
OD
```

A short interlude about performance and automata theory: In a classic paper, Donald E. Knuth introduced a collection of techniques for automatically constructing efficient parsers for the "LR languages", a useful subset of the context free languages [6]. Many tools, including YACC, are based on LR parsing theory or its aantecedent theory of regular expressions and finite automata. These techniques reduce the inner loop of a language translator to a single state machine step plus (in the LR case) a stack operation. Consequently tools based on these theories are very efficient. However, LR theory doesn't apply in any obvious way to LIM, nor does regular expression parsing theory apply to the regular part of LIM (LIM with procedure calls restricted to tail calls); both because LIM has deterministic choice instead of non-deterministic choice and because LIM commands don't just read input, they also write output. We suspect that it would be possible to modify Knuth's theory so that it applied to a large subset of LIM, and if such a theory were developed, and a LIM implementation built that was based upon it, then commands like the one above would run at the speed of a tightly coded state machine. However, we haven't attempted this. Our interpreter simply saves the state and restores it when backtracking is found to be necessary. Executing the burrows-gnelson swap progam on our 500 megahertz EV6 Alpha, our interpreter

translates input to output at 2.6 megabytes per second.[1] On the same platform, The regular expression-based tool `sed` excutes a very similar program 5.3 times faster than LIM (It would be awkward to write a truly equivalent `sed` program; the `sed` program we wrote performs three "`s/.../g`" commands; it has the identical effect provided that the string "`abc99`" does not appear anywhere in the input.)

In a precise technical sense, `DO-OD` is to the regular expression $*$ operator as deterministic choice is to non-deterministic choice. As would be expected from this observation, `DO-OD` is generally more useful than $*$. Regular-expression based lexical tools wrestle around the non-determinism of $*$ in a variety of ways. For example, Michael Lesk's popular tool LEX imposes a "longest match rule" [11]). The wrestling is mostly avoided if $*$ is replaced by `DO-OD`.

3.5 LIM language definition

We now give a complete definition of the LIM language.

3.5.1 PROGRAMS.

A program is a sequence of global declarations. There are two kinds of global declarations: procedure declarations and global variable declarations. Each of these global declarations is terminated by a semicolon.

The effect of executing a LIM program is to initialize all its global variables (in the order in which they are declared) and then to execute the procedure call `Main()`. This procedure must be declared and must have no parameters.

For example, here is a complete LIM program for counting the number of newlines in a file

```
VAR n := 0;

PROC Main() IS
  DO Rd('\n'); n := n + 1 | EVAL Rd() OD
END;
```

The example is rather useless, since it doesn't print the count. But it does illustrate global declarations.

3.5.2 PROCEDURE DECLARATIONS.

A procedure declaration has the syntax

```
PROC outs := inouts:P(ins) IS B END;
```

where `P` is an identifier, the name of the procedure, `outs`, `inouts`, and `ins` are the formal parameters of various modes, and `B` is a command, the body of the procedure. If there are no out parameters, "`outs :=`" must be omitted. If there are no inout parameters, "`inouts :`" must be omitted. If there are no in parameters, `ins` must

[1] For the much larger LIM program that implements Nanoweb, the speed falls to 406 kilobytes per second.

be empty, but the parentheses are still required. Both `ins` and `outs` are comma-separated lists of variables, but `inouts` is either a single variable, or a parenthesized comma-separated list of at least two variables.

For example,

```
PROC CopyToEof() IS DO Wr(Rd()) OD END;

PROC n:Inc() IS n := n + 1 END;

PROC (x,y):Swap IS
  VAR t := x IN x := y; y := t END
END;

PROC n := Square(m) IS n := m * m END;
```

We find that LIM's in and out parameters are similar in function to the inherited and synthesized attributes of attribute grammers as introduced by Knuth [7], but less esoteric.

3.5.3 GLOBAL VARIABLE DECLARATION

A global variable declaration has the syntax

```
VAR vlist;
```

where `vlist` is a comma-separated list of expressions of the form

```
v := E
```

where `v` is an identifier (the name of the global variable being declared) and `E` is an expression (the initial value for the variable).

3.5.4 STATE SPACE AND VALUE SPACE.

LIM programs operate on a state space that includes input and output, which are streams of ISOLatin-1 characters, as well as program-declared integer variables.

LIM's variables range over the integers only. No, no floating-point numbers (groan). No strings (sob). No ordered pairs or maps (wail). And yes, in making this strict decision we are knee-capping our own brainchild. But ruthlessness in the pursuit of simplicity is no vice.

Lacking booleans, the guard arrow treats 0 as false and any other value as true.

3.5.5 COMMANDS.

Here are the thirteen syntactic types of LIM commands:

`A; B`	Execute A, then execute B.
`P -> A`	Evaluate P; if it is non-zero, execute A (else fail).
`A | B`	Execute A, else execute B if A fails.
`DO A OD`	Repeatedly execute A as long as it succeeds. (More below.)
`TIL A DO B END`	Repeatedly execute B until executing A succeeds. (More below.)
`VAR vlist IN A END`	Introduce local variables `vlist` and execute A. (More below.)
`v := E`	Evaluate the expression E and assign its value to v. Fail if any procedure called by E fails.
`EVAL E`	Evaluate the expression E and discard the result. Fail if any procedure called by E fails.
`SKIP`	A no-op.
`FAIL`	This command always fails.
`ABORT`	Print an error message and abort the program.
`outs := inouts:P(ins)`	Procedure call. (More below.)
`{ A }`	Execute A; braces are just for grouping.

Semicolon binds more tightly than the guard arrow, which binds more tightly than deterministic choice. But `A ; P -> B` means `A ; { P -> B }` since this is the only legal interpretation.

The loop `DO A OD` is defined by unrolling it once:

```
DO A OD  =  A; DO A OD | SKIP
```

It is worth noting that `DO A OD` never fails. For example, `DO FAIL OD` is the same as `SKIP`. It is also worth noticing that the recursion defining `DO`-`OD` differs from the recursion defining the regular expression $*$ operator only in the substitution of deterministic choice for non-deterministic choice.

The loop `TIL A DO B END` is also defined by unrolling it once:

```
TIL A DO B END   =   A   |  B; TIL A DO B END
```

It is worth noting that `TIL A DO B END` will fail if A and B both fail after some number (possibly zero) of iterations of B. It is also worth noticing that A may consist of several different cases connected with the deterministic choice operator, in which case the loop has multiple exits.

The command

```
VAR v := E IN A END
```

introduces a local variable v and then executes `v := E; A`. The scope of v includes E and A; it is probably a mistake (undetected by LIM) if v occurs in E.

For example, the following procedure reads a decimal digit and returns its value, or fails if the next input character is not a digit:

```
PROC n := DigitVal() IS
  VAR c := Rd() IN
      '0' <= c AND c <= '9' -> n := c - '0'
  END
END
```

Just as in a global variable declaration, a `vlist` can be a comma-separated list of expressions of the form `v := E`. For example

```
VAR v1 := E1, v2 := E2 IN A END
```

is short for

```
VAR v1 := E1 IN VAR v2 := E2 IN A END END.
```

A procedure call command has the form

```
outs := inouts:P(ins)
```

where `P` is a procedure and `outs`, `inouts`, and `ins` specify the actual parameters of the various modes. If there are no out parameters, "`outs :=`" must be omitted. If there are no inout parameters, "`inouts :`" must be omitted. If there are no `in` parameters, `ins` must be empty, but the parentheses are still required. Syntactically, `ins` is a comma- separated list of expressions, `outs` is a comma-separated list of identifiers, and `inouts` is either a single identifier or a parenthesized comma-separated list of at least two identifiers. A procedure call is allowed to appear earlier in the program text than the procedure's declaration.

To execute the call, the procedure's in and inout formals are bound to the values of the corresponding actuals; then the body of the procedure is executed, and finally the out and inout actuals are bound to the values of the corresponding formals.

3.5.6 EXPRESSIONS.

Here are the operators allowed in LIM expressions, listed in order of increasing binding power.

`e OR f`	logical disjunction; `f` is not evaluated if `e` is non-zero
`e AND f`	logical conjunction; `f` is not evaluated if `e` is zero
`NOT e`	logical negation
`e = f`	equals
`e # f`	differs
`e < f`	less than
`e > f`	greater than
`e <= f`	at most
`e >= f`	at least
`e + f`	sum
`e * f`	product
`e DIV f`	the floor of the real quotient of `e` and `f`
`e MOD f`	`e - f * (e DIV f)`
`-e`	unary minus

The boolean operations produce 1 for true, 0 for false.

Expressions can also have the forms:

`inouts:P(ins)`	procedure call
`v`	Value of the variable `v`
`literal`	An integer, string, or character literal
`(e)`	round parentheses for grouping in expressions

A procedure can be called in an expression if it has exactly one out parameter. The syntactic rule for `inouts` and for `ins` are the same as those previously described for a procedure call command, as are the rules for binding the `in` and `inout` parameters. However, for a procedure call in an expression, no actual `out` parameter is supplied, and after executing the body of the procedure, the value of its formal `out` parameter becomes the value of the expression.

An integer literal is a non-empty sequence of decimal digits; it is interpreted base ten.

A character literal is a single printing character or escape sequence enclosed in single quotes. It denotes the integer ISOLatin-1 code for the enclosed character. The escape sequences allowed are

`\\`	backslash (\)
`\t`	tab
`\n`	newline
`\f`	form feed
`\r`	return
`\s`	blank space
`\b`	backspace
`\v`	vertical tab
`\e`	escape
`\'`	single quote
`\"`	double quote
`\ddd`	char with octal code `ddd`
`\dd`	char with octal code `dd`
`\d`	char with octal code `d`
`\xhh`	char with hex code `hh`
`\xh`	char with hex code `h`

A string literal is a sequence of printing characters or escape sequences surrounded by double-quotes. A string literal may not span multiple lines of the LIM source program. String literals are allowed only as parameters to the built-in procedures `Rd`, `Wr`, `At`, and `Err`: they are not allowed in general expressions.

3.5.7 IDENTIFIERS, COMMENTS, AND TOKEN SEPERATION.

Variables and procedures are denoted by identifiers that consist of a sequence of letters, digits, and underbars beginning with a letter or underbar.

LIM tokens can be separated by white space and comments. A comment is an arbitrary sequence of characters opened by `(*` and closed by `*)`. Comments nest.

3.5.8 BUILT-IN PROCEDURES.

LIM provides the following built-in procedures:

`Rd(str)`	Read `str` (a string literal), or fail.
`Rd(c)`	Read the character `c` (an expression), or fail.
`Rd()`	Read and return the next character, or fail.
`Wr(str)`	Write `str` (a string literal).
`Wr(c)`	Write the character `c` (an expression).
`At(str)`	A noop if `str` (a string literal) is next in the input; otherwise fails.
`At(c)`	A noop if the next character is `c`; otherwise fails.
`Eof()`	A noop if at end of file; else fails.
`Err(str)`	Write `str` to standard error.
`Err(c)`	Write the character `c` (an expression) to standard error.

The primitive `Err` is useful for debugging, but it does not obey the rules of LIM: the output that it performs will not generally be undone, even if the computation fails. For example,

```
Err("A"); FAIL
```

is NOT equivalent to `FAIL`; it will print "`A`" to standard error as it backtracks. The exact output of a program containing calls to `Err()` depends on which internal optimizations the LIM interpreter uses. But you can rely on the idiom `Err(<message>); ABORT` to print the error message and abort the LIM program.

3.5.9 ERRORS.

LIM reports syntax errors or other static language violations with a line number. Don't take the line number too seriously. In this case the interpreter exits with Unix exit code 2.

Common syntax errors are to include a redundant trailing semicolon (within commands, LIM uses semicolon to denote sequential composition, not to end a statement) or to omit the required empty parentheses on a procedure with no in parameters.

If the LIM program aborts, the interpreter prints the line number at which it aborted and the number of characters the program had read and written at the time it aborted, and exits with code 2.

If the LIM program halts normally, the interpreter exits with code 0.

If the procedure `Main()` fails, LIM prints the error message "`guard failed`", together with the "high-water mark", which is the maximum number of characters that the program read from the input at any time in the backtracking execution before it was determined to fail; then the interpreter exits with code 1. Often an examination of the input in the vicinity of the high-water mark for reading can help you detect the reason for the failure. If you are writing a parser in LIM, you may want make sure this never happens, for example, by replacing some procedure body `B` with `B | <recovery action>)`. If the recovery action is simply to print an error message and the procedure body is a outer procedure like `Main`, then this transformation is

trivial to perform. If you want to try more ambitious recovery actions from lower-level procedures, then the transformation can be more work.

A convenient way to reverse engineer a file format is to encode the current hypothsis as a LIM program, test the hypothesis by applying the program to sample files, and, in case of failure, refine the hypothesis by examining the actual file contents in the vicinity of the high-water mark.

3.5.10 LIM EXECUTABLE FILES.

The interpreter ignores the first line of its input file if it starts with the character "#". This allows the UNIX kernel to invoke the LIM interpreter on LIM programs if the file has its execute bit set and starts with a line of the form: `#!lim_pathname` where `lim\_pathname` is the pathname of the LIM interpreter. You can find this pathname by typing "which lim" to your shell.

3.5.11 FURTHER HINTS AND EXAMPLES.

When using LIM to parse context-free syntax, it is necessary to left-factor the grammar, both to avoid grammatical left recursion (which would create infinite recursion in LIM), and, usually, to reflect the different binding powers of the different operators. For example, here is a program that parses and evaluates an arithmetic expression containing sums, differences, products, decimal numerals, and parenthesized subexpressions:

```
PROC Main() IS WriteInt(Eval()) END;

(* "Eval()" (parses and) evaluates an expression. *)

PROC n := Eval() IS
    n := Eval1();
    DO Token('+'); n := n + Eval1()
     | Token('-'); n := n - Eval1()
    OD
END;

(* "Eval1()" evaluates an expression
   tight enough to be an argument to "+". *)

PROC n := Eval1() IS
    n := Eval2();
    DO Token('*'); n := n * Eval2() OD
END;

(* "Eval2()" evaluates an expression
   tight enough to be an argument to "*". *)

PROC n := Eval2() IS
```

```
    n := RdInt()
  | Token('('); n := Eval(); Token(')')
END;

PROC n := RdInt() IS
    SkipWhite();
    n := RdDigit();
    DO n := 10 * n + RdDigit() OD
END;

PROC n := RdDigit() IS
    VAR c := Rd() IN
      '0' <= c AND c <= '9' -> n := c - '0'
    END
END;

(* "Token(c)" skips white space then
    reads the single-character token "c". *)

PROC Token(c) IS SkipWhite(); Rd(c) END;

PROC SkipWhite() IS DO Rd(' ') | Rd('\t') OD END;

PROC WriteInt(n) IS
    n = 0 -> Wr('0')
  | n < 0 -> Wr('-'); WritePos(-n)
  | WritePos(n)
END;

PROC WritePos(n) IS
    n = 0 -> SKIP
  | WritePos(n DIV 10); Wr('0' + (n MOD 10))
 END;
```

A common error is to forget to properly sort the cases of a long syntactic category. For example, consider something like the following fragment of code, which attempts to read identifiers, integers, or function applications:

```
    Id()
  | Int()
  | Id(); Token('(');
    { Token(')')
    | Expr();
      TIL Token(')') DO Token(','); Expr() END;
    }
   ...
```

This is a mistake: the function application case will never be taken, since whenever it could be, the first case will be taken instead. The mistake can be fixed by reordering the cases.

3.6 Writing Fast LIM Programs.

The basic strategy of the current implementation is to save the state before executing `A | B`; if `A` fails, the implementation restores the old state before continuing with `B`. Similarly, the state must be saved before each iteration of a DO loop or TIL loop. However, the implementation omits saving the state if static analysis shows that it is unnecessary. For example, to execute

```
Wr('0'); Rd('0') | ...
```

the state must be saved, since it may be necessary to undo the write, but to execute

```
Rd('0'); Wr('0') | ...
```

there is no need to save the state, since if the left argument to `|` fails, it does so before causing any side effects. As a result, the second command executes somewhat faster than the first.

3.7 Bugs.

The interpreter does not detect arithmetic overflow in the program it is interpreting. It doesn't detect stack overflow, either, although your operating system might.

3.8 See also

`pplim`, distributed with LIM, is a pretty-printer for LIM that is itself written in LIM. In addition to being an instructive example of LIM programming, it provides a template that can easily be instantiated to produce pretty-printers for other languages.

3.9 Authors

Mike Burrows and Greg Nelson designed LIM, implemented LIM, and wrote this man page.

4 The Nanoweb program

This section presents the implementation of Nanoweb, which is a Lim program. At the risk of being somewhat subjective, I would claim that the implementation of Nanoweb in Lim resembles a big grammar more than it resembles a conventional parser. To the extent that grammars are judged to be declarative and manageable, the same might be said of Lim programs.

Of course, this section is also an illustration of what ultralight literate programs look like after they have been typeset.

This is the source code for Nanoweb, typeset by Nanoweb itself.

4.1 The Main procedure

The procedure `Main` translates the entire source.

```
PROC Main() IS (* delta = 0 *)
  SkipToBlankLine(); (* delta = 1 *)
  DO BlankLine(0) OD;
  SetBPFlag();
  Prolog(); (* delta = 0 *)
  MainLoop();
  Epilog()
END;
```

In the comments, `delta` is the number of newlines read less the number written. Its value is commented whenever it changes.

`Main` skips to the first blank line (by calling `SkipToBlankLine`, defined below). (`SkipToBlankLine` also increments `delta`.) The loop `DO BlankLine(0) OD` skips any additional blank lines that immediately follow the first. Then `Main` calls `SetBPFlag`, which sets or clears a global flag controlling whether to print LaTeX boilerplate. This flag is read by `Prolog` and `Epilog`, which write any necessary boilerplate (`Prolog` also decrements `delta`.) Bracketed by the calls to `Prolog` and `Epilog` is the call to `MainLoop` that does the real work:

```
PROC MainLoop() IS
  TIL Eof() | XCom(); MainLoop() DO
    CodeLine()
  END
END;
```

This loop repeatedly calls `CodeLine`, which translates a line of code in code mode. Eventually, the code lines end, either at the end of the file, in which case there is nothing more to do, and `MainLoop` exits, or with an exposition comment, in which case `MainLoop` calls `XCom()` to translate that comment in comment mode, and then recursively calls `MainLoop` to translate the rest of the raw source.

(The Denali-2 project uses the ESC/Java tool [4] to check its source code, whose annotations look best when typeset in code mode. Therefore, in the version of `nanoweb` used by the Denali project, `MainLoop` is slightly different. The changes are very minor. The `nanoweb` tool is released with the expectation that users will tailor it to meet their needs in similar ways.)

4.2 Miscellaneous global declarations

The flag `bpflag` controls whether LaTeX boilerplate is to be produced. It is initialized to one (LIM for **true**):

```
VAR bpflag := 1;
```

`SetBPFlag()` reads the one-word comment `noboilerplate` and clears `bpflag`, otherwise it is a noop.

```
PROC SetBPFlag() IS
    RdOpen();
    Rd("noboilerplate");
    RdClose();
    bpflag:= 0
  | SKIP
END;
```

`RdOpen`, `RdClose`, `WrOpen`, and `WrClose` read and write comment delimiters. Their bodies are trivial, they are made into procedures to facilitate retargeting Nanoweb to languages with different comment delimiters. For example, in the version of `nanoweb` used to typeset itself, these procedures were changed to read and write openpren-star and star-closepren (the LIM delimiters) instead of the Java/C slash-star and star-slash.

```
PROC RdOpen() IS Rd("/*") END;
PROC WrOpen() IS Wr("/*") END;
PROC RdClose() IS Rd("*/") END;
PROC WrClose() IS Wr("*/") END;
```

`SkipToBlankLine` skips up to and including the next blank line. It consumes the blank line without emitting a compensating newline; that is, it increments `delta` (because the compensating newline will be written by `Prolog`).

```
PROC SkipToBlankLine() IS
  TIL BlankLine(1)
    | Eof(); Err("nanoweb: no blank line found"); ABORT
   DO ReadLine()
  END
END;
```

The call `BlankLine(b)` reads a blank line (or fails if the next characters in the input don't form a blank line). It also emits a newline unless `b` is true, in which case it emits nothing. Thus for one-bit `b`, `BlankLine(b)` has the effect `delta := delta + b`. `ReadLine()` reads and discards the next line of input. It never fails

```
PROC BlankLine(b) IS
  TIL Rd('\n') DO Rd(' ') | Rd('\t') END;
  { b -> SKIP | Wr('\n') }
END;


PROC ReadLine() IS
  TIL Rd("\n"); Wr("\n") | Eof() DO EVAL Rd() END
END;
```

`BlankLine(0)` and `ReadLine()` write newlines in order to preserve line numbers, so that TeX error messages relative to Nanoweb output will be accurate relative to its input as well. `BlankLine(1)` is useful to read the first blank line of the raw source, whose compensating newline will be included with the LaTeX boilerplate.

The next procedures, `Prolog` and `Epilog`, write the LaTeX boilerplate (unless it is suppressed). `Prolog`'s output contains exactly one newline, regardless of `bpflag`, so that it decrements `delta`. When we get to the epilog, we no longer care about `delta`.

```
PROC Prolog() IS
  bpflag ->
    Wr("\\documentclass[12pt]{article}");
    Wr("\\usepackage{times}");
    Wr("\\usepackage{mathtime}");
    Wr("\\begin{document}\n")
  | Wr("\n")
END;


PROC Epilog() IS
  bpflag -> Wr("\n\\end{document}\n") | SKIP
END;
```

4.3 code mode

`CodeLine()` translates one line in code mode, starting from the beginning of the line.

```
PROC CodeLine() IS
    BlankLine(1);
    Wr("\\par\\penalty-500\\medskip\\noindent\n")
  | Wr("\\mbox{\\tt ");
    TIL {Rd('\n') | Eof()}; Wr("}\\\\\n")
     DO CodeElem()
```

```
    END
END;
```

A blank line in the code becomes an explicit paragraph boundary together with a medium vertical skip. The negative penalty command indicates a good place for a page break.

A non-blank line is translated into "`\mbox{...}\\`" where "`...`" is the sequence of translations of the characters on the line (as provided by `CodeElem`). (Warning: "`mbox`" and "`\\`" are LaTeXisms. If `nanoweb` is used with other TeX macro packages, they must be changed.)

`CodeElem()` translates a single element of the code. This is typically a single character to be echoed from input to output. But several special cases arise. (1) If the character in the code is special to TeX, it must be quoted (with `QuoteTexChar`). (2) Blank spaces in code should become TeX control spaces, since two are not the same as one. (3) A single-line-comment in code mode is read and translated by a `TIL` loop as one of the cases of a `CodeElem`, since a single-line code comment is supposed to be typeset as code, (4) A string literal in code mode is also read and translated by a `TIL` loop as one of the cases of a `CodeElem`, to prevent an open comment delimiter in a string literal from being mistaken as a real open comment. (5) A character literal is also translated as a single `CodeElem`, in case it is a double-quote. This produces:

```
PROC CodeElem() IS
    QuoteTexChar()
  | Rd(' '); Wr("\\ ")
  | RdOpen(); WrOpen();
    TIL RdClose(); WrClose()
      | {At('\n') | Eof()};
        Err("nanoweb:  illegal code comment");
        ABORT
     DO CodeElem()
    END
  | Echo('\"'); TIL Echo('\"') DO CharOrEscape() END
  | Echo('\''); CharOrEscape(); Echo('\'')
  | Wr(Rd())
END;
```

`CharOrEscape` translates a single character or string literal escape sequence:

```
PROC CharOrEscape() IS
  Rd('\\'); Wr("\\char\'134{}"); (* Echo a backslash. *)
  { Echo('n') | Echo('t') | Echo('b')
  | Echo('e') | Echo('r') | Echo('f')
  | Echo('\'') | Echo('\"')
```

```
  | Rd('\\'); Wr("\\char\'134{}")
  | OctDigit(); {OctDigit() | SKIP}; {OctDigit() | SKIP}
  | Echo('x'); HexDigit(); {HexDigit() | SKIP}
  | Echo('u'); HexDigit(); HexDigit(); HexDigit(); HexDigit() }
| QuoteTexChar()
| Rd(' '); Wr("\\ ")
| Wr(Rd())
END;
```

Whew. That was a slog. But it is easy coasting from here:

`Echo` reads a specified character from the input and writes that same character to output:

```
PROC Echo(c) IS Rd(c); Wr(c) END;
```

`OctDigit` and `HexDigit` echo any octal digit, or any hexadecimal digit, respectively. They rely on `Echo2(lo, hi)`, which echoes any character within the range `[lo..hi]`.

```
PROC OctDigit() IS Echo2('0', '7') END;

PROC HexDigit() IS
  Echo2('0', '9') | Echo2('a', 'f') | Echo2('A', 'F')
END;

PROC Echo2(lo, hi) IS
  VAR c := Rd() IN
    lo <= c AND c <= hi -> Wr(c)
  END
END;
```

`QuoteTexChar()` reads a character that has special significance to TeX and emits a `\char` command that will typeset that character without confusing TeX. We don't want this `\char` command to cause TeX to ignore any blank spaces that might be next, so we insert an empty group (`{}`) after it:

```
PROC QuoteTexChar() IS
    VAR ch := Rd() IN
      ch = '%' OR ch = '_' OR ch = '&' OR ch = '$' OR
        ch = '^' OR ch = '#' OR ch = '{' OR ch = '}' OR
        ch = '\\' OR ch = '~' ->
      Wr("\\char\'"); OutOct(ch); Wr("{}")
    END
END;
```

For a natural number `n`, `OutOct(n)` writes `n` in octal if it is positive, and writes the empty string if `n` is zero. Since the character with code zero is not one of the TeX special characters, the first case occurs only in the recursive call, not in the call from `QuoteTexChar`:

```
PROC OutOct(n) IS
  n = 0 -> SKIP | OutOct(n DIV 8); Wr('0' + (n MOD 8))
END;
```

4.4 Comment mode

`XCom()` translates an exposition comment, starting from the beginning of the line on which the comment is opened.

```
PROC XCom() IS
  TIL RdOpen() DO Rd(' ') | Rd('\t') END;
  TIL RdClose()
    | Eof(); Err("nanoweb: unclosed comment"); ABORT
   DO ComElem() END
END;
```

First `XCom` skips blank space and reads the open comment delimiter. Then it translates the remainder of the comment by repeatedly translating an "element of the comment" with `ComElem`.

```
PROC ComElem() IS
    Rd('\n');
    BlankLine(1);
    Wr("\n\\par\\medskip\\noindent\n")
  | Rd("\"");
    Wr("\\leavevmode\\hbox{\\tt ");
    TIL Rd("\""); Wr("}") DO
      Rd('\n');
      Err("nanoweb: newline in code escape");
      ABORT
    | CodeElem()
    END
  | Wr(Rd())
END;
```

An element of a comment is either a blank line to be translated into a medium vertical skip, a double-quoted phrase to be translated in code mode (by repeated calls to `CodeElem`), or a character to be copied from input to output. (Neither Java nor C allow nested comments. In languages like Modula-3 and LIM that do allow nested comments, a fourth alternative should be added to `ComElem`. For example, adding the alternative `| RdOpen(); TIL RdClose() DO ComElem() END` would simply flatten nests of comments by inlining each nested comment into the containing comment.) The `leavevmode` command before the `\hbox` is important in case a double-quoted phrase appears as the initial phrase in a paragraph, as explained in Donald E. Knuth's *TeXBook* [9].

Fini.

5 Concluding remarks

Just as with other literary genres, it is more work to write a short program than a long one. As I write these sentences, I have just finished two weeks of hard work writing Nanoweb. In the end, I met my budget with more than ninety lines to spare, as revealed by a final `wc nanoweb.lim nanoweb.1 test/*.in`.

During those two weeks, I also revised the LIM man page and wrote the first draft of this paper, but even considering this, 410 lines in two weeks is much much slower than I write code when I am not under such a tight complexity budget. Some managers may be horrified by what they mistakenly think to be a drastic reduction of productivity. I would remind them that lines of code represent cost, not value. Experienced programmers don't need this remindeer, they know it all too well; I would remind them that though the work required to write short programs is difficult, it is also satisfying.

Finally: I hope, gentle reader, that you will find LIM and Nanoweb to be useful, and that you have also found them to be a good read.

References

[1] Alfred V. Aho and Jeffrey D. Ullman. *Principles of Compiler Design*. Addison-Wesley, 1979.

[2] Mike Burrows and Greg Nelson. Lim source code. Currently available via `www.research.compaq.com/src/lim`. If that page moves (for example, into the HP Labs web server), try Google, 1993. As of 2005, the LIM download page includes the code for Nanoweb.

[3] Edsger W. Dijkstra. *Discipline of Programming*. Prentice-Hall, 1976.

[4] Cormac Flanagan, Rustan Leino, Mark Lillibridge, Greg Nelson, and Raymie Stata. Extended static checking for Java. *Proceedings of the ACM PLDI Conference*, July 2002.

[5] Rajeev Joshi, Greg Nelson, and Yunhong Zhou. The straight-line automatic programming problem. Technical Report HPL-2003-236, HP Laboratories Palo Alto, November 2003. To appear in TOPLAS.

[6] D. E. Knuth. On the translation of languages from left to right. *Information and Control*, 8(6), 1965.

[7] D. E. Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, 2(2), 1968.

[8] D. E. Knuth. Literate programming. *The Computer Journal*, 27, 1984. Reprinted in [10].

[9] Donald E. Knuth. *The TEXbook*. Published jointly by the American Mathematical Society and the Addison-Wesley Publishing Company, 1986.

[10] Donald E. Knuth. *Literate Programming*. Center for the Study of Language and Information, Leland Stanford Junior University, 1992.

[11] M.E. Lesk. LEX—a lexical analyser. Research Report CSTR 39, Bell Labs Computer Science Center, 1975. Cited by Aho and Ullman [1].

[12] Greg Nelson. A generalization of Dijkstra's calculus. *ACM Transactions on Programming Languages and Systems*, October 1989.

[13] Greg Nelson. Some generalizations and applications of Dijkstra's calculus of guarded commands. In Manfred Broy, editor, *Programming and Mathematical Method*, volume 88 of *NATO Advanced Science Institute Series F (Computer and Systems Sciences)*, pages 157–91. Springer Verlag, 1992.

[14] Norman Ramsey. Literate programming simplified. *IEEE Software*, 11(5), September 1994.