ELECTRONIC PUBLISHING, VOL . 2(3), 157–167 (OCTOBER 1989)

INSCRIPT — a C-like preprocessor for POSTSCRIPT*

JAKOB GONCZAROWSKI
*Typographics,Ltd.***
Jerusalem, Israel

ON G. PARADISE
Computer Science Department
The Hebrew University of Jerusalem
Jerusalem 91904, Israel

SUMMARY

INSCRIPT is a front-end for the POSTSCRIPT page-description language. INSCRIPT is easier to write (and read) than POSTSCRIPT as it uses high-level syntax, performs automatic stack manipulation and defines a clearer interface to the POSTSCRIPT imaging model. INSCRIPT programs for graphic imaging can be developed interactively, or compiled to produce POSTSCRIPT code for off-line use.

This paper describes the INSCRIPT environment, its language features, its implementation, and the way POSTSCRIPT code is generated from its various constructs.

Possible enhancements to POSTSCRIPT are suggested which would turn it into a better 'execute engine' for code generated from high level languages. Direct POSTSCRIPT programming would then be much easier as well.

KEY WORDS High-level language interface POSTSCRIPT Program readability Stack languages Variable allocation

INTRODUCTION

The POSTSCRIPT™ page-description language is a popular interface to high-quality printers, typesetters and CRT displays. It has a comprehensive set of graphic and typesetting operators, called the *imaging model*, along with some general-purpose programming constructs.

This document overviews INSCRIPT — a tool for interactive development of software to produce graphic images. INSCRIPT is a 'pleasant' programming dialect patterned after the C programming language. A compiler translates INSCRIPT into POSTSCRIPT code that can be executed on graphic output devices. Typing a program into the compiler results in the immediate display of a graphic image, which aids program debugging and tuning.

Since POSTSCRIPT was not designed for such an environment, the performance of code produced by INSCRIPT leaves a lot to be desired. This may be solved by minor enhancements to POSTSCRIPT, which are discussed in detail.

* POSTSCRIPT is a registered trademark of Adobe Systems Inc.
** While on leave from the Hebrew University of Jerusalem.

0894–3982/89/030157–11$05.50
Received 22 February 1989
Revised 17 November 1989

Motivation

POSTSCRIPT was not designed for human programmers. The language reference manual [1] states:

> Normally, POSTSCRIPT page descriptions are generated automatically by composition programs such as word processors, illustrators, computer aided design systems, and others. Programmers generally write POSTSCRIPT programs only when creating new applications. However, in special situations a programmer may write POSTSCRIPT programs to take advantage of POSTSCRIPT capabilities that are not accessible through a particular application program.

Typical typesetting applications use several hundred lines of handcrafted POSTSCRIPT code. However, since POSTSCRIPT is powerful and general, it has been adapted to describe user interfaces in distributed windowing environments. These applications send a POSTSCRIPT program to a display server, which executes it to obtain the desired display image. A user-interface environment [2] may require ten thousand lines of POSTSCRIPT code.

The authors' experience shows that after getting used to the postfix notation of the language, POSTSCRIPT programs are not too hard to write provided there is little need for accessing 'variables' (which involves very tedious counting of the elements presently on top of the stack). On the other hand, even small programs are very hard to read, debug, and maintain. POSTSCRIPT interpreters use many stacks (for operands, control, naming, and graphics) which must be explicitly maintained by the programmer. Since the graphic representation of a program (indentation) is one-dimensional, some of the information about the state of these stacks cannot be conveyed to the reader of a POSTSCRIPT program. The postfix notation makes things even less pleasant.

Similar problems have been solved by Kernighan's Ratfor preprocessor [3] for Fortran:

> When one is faced with an unpleasant language, a useful technique is to define a new language that overcomes the deficiencies, and to translate it into the unpleasant one using a preprocessor.

If the 'new language' is too radical, few people will take the time and effort to learn it. Therefore, INSCRIPT is patterned after the popular C programming language [4]. C uses high-level syntax to manipulate low-level objects; POSTSCRIPT takes things to the other extreme by using primitive syntax to handle high-level objects (e.g. fonts). INSCRIPT is an attempt to create a more balanced language by picking the best from both.

THE INSCRIPT ENVIRONMENT

Although C is usually compiled, its syntax is suitable for interpreted languages [5, 6] for interactive use. The INSCRIPT compiler produces code on the fly, so it can be used interactively as well. This interactivity is of crucial importance for two reasons. First, programming to produce graphic output requires considerable tuning, which is easier to do interactively. Second, the semantics of POSTSCRIPT imply that some programming errors can only be detected at run-time. Since all POSTSCRIPT interpreters perform extensive run-time checking, INSCRIPT relies on this mechanism for error handling.

In a typical interactive setup(Figure 1), the INSCRIPT compiler reads its input from a

keyboard. As the compiler output is generated on the fly, routing it to a POSTSCRIPT interpreter results in immediate execution. If the interpreter produces graphic images on a CRT, they can be viewed as the INSCRIPT code is typed in; otherwise, images can be viewed by invoking the POSTSCRIPT showpage operator[1].

When working interactively, one would create INSCRIPT program files with a text editor. Using the `#include` directive of the preprocessor, these files are then interactively loaded along with library code. Then, the programmer may type in short sequences of code (e.g. call a function, modify parameter values) and examine the resulting image. When the program is to be modified, it can be edited and `#included` again to supersede older versions.

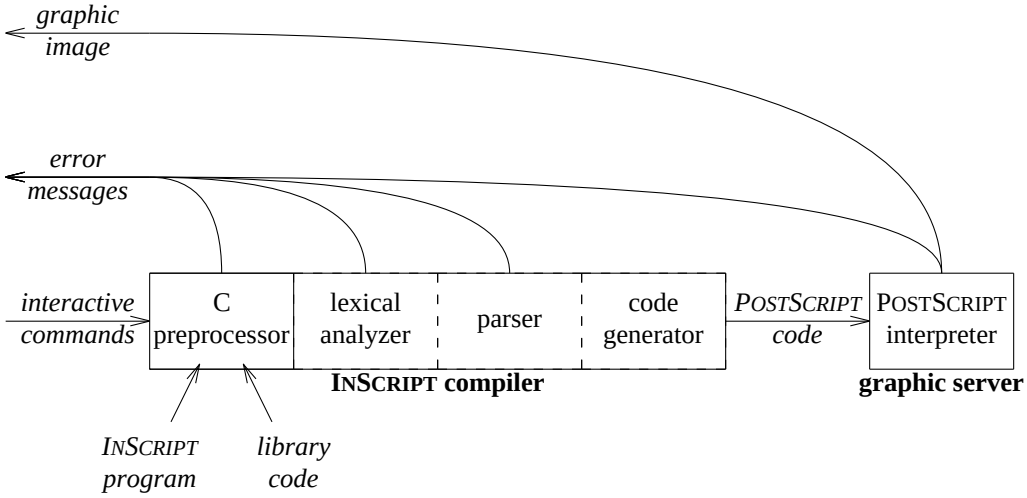

Figure 1. An interactive INSCRIPT development setup

After a stable version of the program is ready, it can be compiled into POSTSCRIPT and saved; INSCRIPT need not be invoked again unless the program is modified.

Binding and typing

Like many other interpreted languages, POSTSCRIPT has no variables. Instead, it provides means for dynamic binding and typing: objects can be named at run-time, and names can be used later to retrieve the objects to which they refer. Types are attributes of the objects themselves; no names are typed, and any object can be bound to any name.

Dynamic binding and typing require no manifest information (declarations). Hence, POSTSCRIPT program fragments can be generated with little or no reference to the context in which they execute. Application programs take advantage of this as they generate code fragments on a host computer and down-load them incrementally into remote POSTSCRIPT servers, such as printers.

The dynamic scheme has, however, two major drawbacks. First, POSTSCRIPT interpreters must look up names at run-time, which slows down program execution. INSCRIPT makes no attempt to solve this inherent POSTSCRIPT problem. Second, interpreters must

[1] Throughout this paper, POSTSCRIPT codes are set in Helvetica and INSCRIPT codes in `Courier`.

detect and report [7] violations of program security (*e.g.*, type clashes) at run-time. Although this entails only a small additional overhead, it makes debugging more difficult: errors are reported after the program has been parsed, so some of the information from the source code is essentially lost. For example, POSTSCRIPT operators check the type of their operands and print a typecheck error message whenever a clash is detected. Unfortunately, these error messages contain only the name of the *operator* that detected the clash; this is of very little help to the programmer, who must know which *operand* is of the wrong type.

INSCRIPT overcomes this difficulty while preserving the dynamic binding and typing scheme of POSTSCRIPT. INSCRIPT's global variables are typeless and need not be declared: they simply come to life when being assigned to. At one time, the authors had considered adding compile-time typing to INSCRIPT. There was a strong reason for not doing this, quite apart from the dynamic type checking facility of POSTSCRIPT. POSTSCRIPT has a multitude of operators, with lots of overloading (operators having different meanings depending on the operand types). To declare each operator in all of its typing contexts would yield a set of declarations so large as to be very unwieldy. INSCRIPT should also not depend upon the particular POSTSCRIPT version at hand, with its particular set of operators, but should be flexible. The reader is referred to the sections 'Operators' and 'Other POSTSCRIPT operators' below.

As all POSTSCRIPT interpreters detect errors at run-time, INSCRIPT relies on this to maintain program security. For debugging, INSCRIPT extends the POSTSCRIPT error handler to provide clearer error messages: whenever an error occurs, POSTSCRIPT will also print the INSCRIPT file name and line number which caused the trouble. Although this extension involves some run-time overhead, it provides a 'compile, check and execute' environment: redirecting the output of INSCRIPT to a POSTSCRIPT interpreter generates detailed error messages of sufficient clarity.

Aggregate data types

INSCRIPT supports the basic data types integer, float and logical. In addition, INSCRIPT supports POSTSCRIPT string, array and dictionary data types in the following way. Elements of arrays and strings can be accessed as

name`[`*expr*`]`

where *name* is an array (or string) and *expr* must have a numeric value (otherwise, the POSTSCRIPT interpreter will invoke a typecheck error). Array elements can be of any type, including other arrays. Thus, multi-dimensional arrays can be created at run-time and accessed as in

name`[`*expr1*`][`*expr2*`]`

As in C and POSTSCRIPT, array elements are numbered starting with 0. Since array need not be declared, they are constructed and initialized at run-time; the function-like construct

`[](`*element_0*`,` *element_1*`, ...,` *element_n*`)`

creates an initialized array of $n+1$ elements which contains *element_0* through *element_n*. Empty arrays (containing null elements) can be created as well.

POSTSCRIPT dictionaries are analogous to records (in Pascal) or structures (in C), except that names of dictionary members are defined at run-time and need not be previously declared. As with arrays, dictionary members can be other dictionaries, so constructs like

dictionary`.`*name1*`.`*name2*`...`

can be used to access a hierarchy of nested dictionaries.

Operators

POSTSCRIPT has a complete set of arithmetic, relational, logical and bitwise operators. INSCRIPT uses them to translate infix expressions into the corresponding POSTSCRIPT code:

```
2 + 3 * 5
```

becomes

```
2 3 5 mul add
```

However, INSCRIPT does not attempt to cover up for slight semantic differences between C and POSTSCRIPT. For example, INSCRIPT translates its negation operator `!` into POSTSCRIPT `not` (which requires a Boolean operand), while C compares the operand of `!` against zero.

There are only a few exceptions in which the semantic difference between C and POSTSCRIPT is too dangerous to ignore. Notably, POSTSCRIPT evaluates logical expressions in full, while C computes them up to the point where the result can be determined (for example, `a && b()` invokes `b()` only if `a` is logically true). In such cases, INSCRIPT generates POSTSCRIPT code that follows the original semantics of C.

POSTSCRIPT has several ways to associate values with names. INSCRIPT avoids this by using a C-like assignment operator. This is 'just another operator' that groups from right to left, thus

```
d.n = a = 3
```

associates the same value with both names. Note that this operator corresponds to one of several POSTSCRIPT operators: `def`, `store` and `put`, depending on the entity to which the assignment is made. In addition, INSCRIPT provides auto-assignment and auto-increment operators (*e.g.*, `+=`, `++`).

Indirect memory access (through pointers) plays a major role in C programming. INSCRIPT provides an equivalent, but less useful, construct to access the name space through names. The unary `*` operator can be applied to *names of names*, thus the expression

```
a = *b + 1
```

makes sense if `b` is a *name of a name* of a number. Unlike C, INSCRIPT cannot determine the name (or address) of an object, so it has no unary & operator. However, the unary `/` operator produces a *literal name* which can be used for indirection, as in

```
c = 2;
b = /c;        /* b  is bound to literal name  c */
a = *b + 1;
```

INSCRIPT operators obey the same precedence rules as in C. These are probably too complicated to remember, so parentheses should be used whenever there is a doubt.

Other POSTSCRIPT operators

There are well over two hundred built-in POSTSCRIPT operators, and new ones are being added as the language evolves. Furthermore, user programs may add (or overload) operators at will. Therefore, it is quite useless to try to teach INSCRIPT about all of them. Instead, INSCRIPT can invoke *any* operator using prefix notation, as if it were a function. Thus the expressions

```
a + 7
add(a, 7)
```

are equivalent.

This scheme provides access to the POSTSCRIPT graphic and font machinery. Unfortunately, it is well-defined only for 'well-behaved' operators that pop their arguments off the stack and push one return-value. INSCRIPT attempts to cover up for other operators, with varying degrees of success, as shown below.

Void operators (with no return value) are easy to handle. To preserve stack alignment, INSCRIPT detects them and acts as if their return value was null, so expressions like

```
1 + print("hello")  /* print() returns nothing */
```

will trigger a typecheck error.

POSTSCRIPT operators that leave multiple return values on the operand stack are harder to handle. For those, INSCRIPT ignores all but the last return value, which is not always adequate. Consider the POSTSCRIPT built-in currentrgbcolor operator, which leaves three values on the stack representing the red, green and blue components of the current color. Using prefix notation, the INSCRIPT expression

```
b = currentrgbcolor()
```

assigns the current blue component to b, but the red and green components are lost.

The INSCRIPT run-time library solves this problem by providing a set of functions that extract one color component at a time. These functions are grouped in a dictionary to provide a clearer semantic than this of multiple-valued operators, as in

```
red = currentcolor.r()
```

The INSCRIPT run-time library replaces POSTSCRIPT operators that return multiple values with functions patterned after the C run-time library. For example, the two-valued input operators of POSTSCRIPT leave the input data on stack, followed by a Boolean value which indicates successful I/O. INSCRIPT standard I/O functions return only the data or an out-of-band value (in case of failure). Modeling this interface after the C standard I/O library enriches INSCRIPT with well-known C programming-idioms such as

```
while ((c = getchar()) != EOF)
       putchar(c);
```

Functions and macros

POSTSCRIPT procedures are quite primitive when compared to other high-level languages. They lack facilities for named arguments and do not support named automatic storage. The responsibility for locating arguments on the stack, removing them and leaving a return value is left to the programmer. This manual stack control is probably the major hardship of POSTSCRIPT programming.

INSCRIPT functions have named variables and automatic storage, so they are inherently re-entrant. Results are handled by a `return` statement, to relieve the programmer of stack-alignment problems.

The syntax of function declaration is the same as in C, except that arguments, automatics and the return value are untyped:

```
f(x, y)
{
        auto a, b = 3;

        a = x + y;
        return(a / b);
}
```

As in C, automatic storage can be allocated at the beginning of any block, using the `auto` keyword[2].

Pointers to functions are not needed in INSCRIPT: once declared, `f` is a name of a function (which may have other names as well). Writing

```
g = f;
```

aliases `g` to the function name `f`, so the two expressions

```
a = f(1, 3)
a = g(1, 3)
```

have the same effect.

The INSCRIPT run-time library uses aliasing to access POSTSCRIPT operators which have peculiar names. For example, the `==` operator (which pretty-prints its operand) is aliased as `pprint` by

```
pprint = *cvn("==");
```

First, the POSTSCRIPT operator `cvn` converts the string `==` into a literal name. Then, the object bound to this name (in this case, the pretty-print function) is associated with `pprint`. This amounts to giving the POSTSCRIPT function another name.

Named arguments and local storage do not come for free: calling a function from INSCRIPT is rather slow, as will be described later. In some cases, in-line macros (created with the preprocessor `#define` directive) are faster.

Implementation

The current version of the INSCRIPT compiler is constructed using conventional UNIX®

[2] The `auto` storage class declaration is optional in C; it is mandatory in INSCRIPT due to syntactic ambiguities.

tools[3]. The lexical analyzer uses lex [8] which is inefficient but adequate. A yacc [9] parser generates control-flow code on the fly, and constructs parse trees for expressions.

We give some details of the compilation technique used. Generating POSTSCRIPT code for expressions requires large amounts of inherited attributes. To overcome this difficulty, INSCRIPT uses parsing [10] to generate code for expressions. Expression trees are scanned in pre-order and fed to a second parser (also constructed with yacc). This amounts to a top-down traverse of expression trees, during which inherited attributes can be easily detected and used to switch sets of code-generating rules. The power of this method is required mainly for auto-assignment operators (`++`, `+=` *etc.*), which may be too complex for other methods to handle. Some optimization is performed by ambiguities in the code-generation grammar, capitalizing on yacc's tendency to shift rather than reduce.

INSCRIPT uses a slightly modified version of the C preprocessor to provide simple run-time library facilities, which consist of header files that may be incorporated into the source code using the `#include` directive. Along with macro facilities, the preprocessor provides other important services. It keeps track of the input file and line number, which are propagated to the POSTSCRIPT `$error` dictionary to report run-time errors. INSCRIPT comments are usually stripped by the preprocessor, but they may be copied by the compiler to annotate the generated POSTSCRIPT code.

POSTSCRIPT AS A TARGET MACHINE

POSTSCRIPT has a rich set of constructs(Table 1) that seem to provide the necessary run-time support for code generated from high-level languages. However, the authors'

Table 1. POSTSCRIPT aids for high-level languages

Sec.*	PostScript primitive	useful for	as in
3.2	operand stack	postfix notation	Forth
3.6	executable array	procedures	Fortran
3.5	operand stack	function	Algol-60
3.6	calling scheme	call by value	Algol-60
3.4	name data type	call by name	Algol-60
3.6	calling scheme	varargs	C
3.8	arithmetic operators	expressions	Fortran
3.4	implicit conversion	expressions	Fortran
3.4	dictionary	dynamic binding	Lisp
3.4	dictionary	record	Pascal
–	dictionary	class	Simula
3.4	dictionary stack	dynamic scope	Lisp
3.4	dictionary stack	class hierarchy	SmallTalk
3.4	exec	eval	Lisp
3.4	stopped/stop	setjmp/longjmp	C library
3.8	control operators	structure	Algol-60
3.4	explicit conversion	cast	C
3.8	type operator	polymorphism	Ada
3.8	type operator	overloading	Ada

* Section number in [1]

[3] ® UNIX is a registered trademark of AT&T in the USA and other countries.

experience shows that most of these services are only conceptually equivalent to those required in practice. INSCRIPT has occasionally to 're-invent' some of them and override others, which results in tricky or slow code.

Fortunately, execution speed (not to be confused with imaging speed) plays only a minor role in POSTSCRIPT (and INSCRIPT) programming: most of the human-produced code executes only a few times, to establish the required setup for imaging. According to Glenn Reid [11], POSTSCRIPT has only limited potential for other uses:

> The fact that POSTSCRIPT is a programming language should not encourage you to perform division or compute arctangents with it. Use it simply to optimize the job of imaging where you can.

Nevertheless, the speed of INSCRIPT code could be dramatically improved by enhancing POSTSCRIPT with a few run-time mechanisms. The following sections describe the authors' wish-list for such enhancements.

Automatic execution (for efficiency)

The POSTSCRIPT automatic execution mechanism is an implicit interface between the operand and the execution stacks, and may affect both of them. Objects pushed onto the operand stack are examined, and if they have an executable attribute (as with functions or operators), they are automatically transferred to the execution stack, which is analogous to function calling.

This mechanism cannot co-exist with block-structured languages: such languages attempt to control both stacks explicitly. Consider the statement `a = b;` at first, it seems as if INSCRIPT could compile it into /a b store. However, if b is a function of one or more arguments, the above code will invoke it and ruin the operand stack alignment. Therefore, INSCRIPT must use /a //b load store which is much slower.

A way to inhibit automatic execution would speed up the code produced by INSCRIPT by more than 30 per cent. This could easily have been accomplished by a global flag.

Stack frame (for efficiency and ease of writing POSTSCRIPT programs)

POSTSCRIPT provides no mechanisms for named function arguments or automatic storage. To overcome this deficiency, INSCRIPT invokes functions via an inefficient calling sequence made of POSTSCRIPT primitives. To demonstrate the penalty in performance, here is a brief synopsis of the sequence:

> First, it computes the arguments on top of the stack.
> Then, it 'shoves' a mark just below the first argument and invokes the called routine.
> This routine pushes a null for each automatic variable, and computes its return value onto the top of the stack.
> After it returns, the calling sequence rolls the stack so everything up to (and including) the mark is cleared, *except* the return value.

This calling sequence is ten times slower[4] than the conventional POSTSCRIPT calling mechanism. Furthermore, arguments and automatic variables are accessed according to their offset from the mark, which involves a counttomark for each fetch. Persons un-

[4] The PDP11 family lacked a stack frame mechanism, so C compilers used a run-time routine for that purpose. Incidentally, this is also an order of magnitude slower than the `jsr` — `ret` instruction pair.

lucky enough to have to write programs in POSTSCRIPT feel this deficiency as well; one has to manually keep track of the stack state for each operation because accessing operands involves counting the location of these from the stack top.

To make things even less efficient, POSTSCRIPT provides no inverse for the index operator: there is no efficient way to store anything into a stack cell. INSCRIPT emulates this operation by two roll instructions. Therefore, using automatic variables in INSCRIPT is highly inefficient, and should be avoided whenever re-entrancy is not essential.

The entailed overhead could be easily waived if POSTSCRIPT provided stack-frame operators: pushframe would establish a new stack frame; popframe would recover an old frame, frame *n* get and any frame *n* put would load and store the *n*th stack element from the frame.

Canonic store **and** load (for simplicity)

POSTSCRIPT uses different operators to store values in the current dictionary and in other aggregates. In practice, a single (polymorphic) operator is sufficient: using the order of operands in Table 2, POSTSCRIPT could determine the required operation according to the type of the topmost stack element.

Table 2. Arguments for suggested unified store operator

operands	meaning
any name dict	name in dictionary
any name	name in dictionary hierarchy
any num array	element of array
any num	offset from stack frame

The suggested reverse-Polish notation could greatly reduce the complexity of INSCRIPT code generation, and would support expressions like

```
a.(b ? c() : d.e)
```

which cannot be compiled now. This is also true for a unified load operator.

Garbage collection (for efficiency and simplicity)

Once an aggregate data type (array, string or dictionary) is allocated, POSTSCRIPT never reclaims the space it occupies. If POSTSCRIPT had a garbage collection mechanism, INSCRIPT and other languages could use dictionaries (instead of stack-frames) for automatic storage and avoid the need for a stack-frame mechanism. Furthermore, dictionary and array operations could become much simpler; in particular, functions could return aggregates rather than many values.

CONCLUSION

POSTSCRIPT was never meant to be used as a back-end for higher-level languages. INSCRIPT attempts to make the best use of what POSTSCRIPT provides, at the risk of being incomplete and inefficient. Despite its deficiencies, the current version of INSCRIPT is much more pleasant to use than raw POSTSCRIPT: it provides a better means for interac-

tive program development, relieves the programmer from manipulating the operands and execution stacks, improves readability via high-level syntax, and simplifies the interface to existing POSTSCRIPT operators and procedures.

Most of the INSCRIPT development effort was spent on design, particularly on what the language should *not* do. For the time being, INSCRIPT provides only those features for which we could clearly define a 'best way' to implement them; some important features were left out, to be incorporated in future versions. Notably, we still have to find a 'best way' to handle classes (neither SmallTalk, C++ nor Ada treat them in the way that INSCRIPT needs). Once classes are incorporated into the language, future versions of INSCRIPT will probably automate the usage of the dictionary stack, much like the current version does for the operand stack.

Acknowledgements

INSCRIPT was designed to fit into the UNIX tool-chest, so it borrows good ideas from other UNIX tools. These include the syntax of the C programming language [4], the binding scheme of the awk pattern scanning language [6], and the structure of the bc interactive bench calculator [5] which pipes the output of a compiler to a stack machine.

A generous loan of equipment from Computer Consoles Inc. (of Ramat-Gan, Israel and Reston, Virginia) made the development of INSCRIPT possible.

This paper owes much to Gwen Hickling's rare combination of (both human and computer) linguistic talents, which provided the authors with many valuable comments.

REFERENCES

1. J. Warnock, D. Brotz, A. Shore, L. Gass, and E. Taft, *POSTSCRIPT Language Reference Manual*, Addison-Wesley, Reading, Mass., 1985. Adobe Systems Inc.
2. J. Gosling, *NeWS*, Sun Microsystems Inc., Mountain View, CA., 1987.
3. B. W. Kernighan, 'Ratfor — a preprocessor for a rational Fortran', *Software — Practice and Experience*, **5** (4), 395–408 (1975).
4. B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, Prentice-Hall, Englewood Cliffs, N.J., 1978.
5. L. Cherry and R. Morris, 'Bc — an arbitrary precision desk calculator', *Unix Programmer's Manual*, Bell Laboratories, Murray Hill, N.J. (1979).
6. A. V. Aho, B. W. Kernighan, and P. J. Weinberger, 'Awk — a pattern scanning and processing language', *Software — Practice and Experience*, **9** (4), 267–280 (1979).
7. B. J. MacLennan, 'Design: objects and message passing', in *Principles of Programming Languages*, Holt-Saunders, New York, N.Y., pp. 467, (1983).
8. M. E. Lesk, 'Lex — a lexical analyzer generator', in *Comp. Sci. Tech. Rep. No. 39*, Bell Laboratories, (October 1975).
9. S. C. Johnson, 'Yacc — Yet Another Compiler-Compiler', in *Comp. Sci. Tech. Rep. No. 32*, Bell Laboratories, (July 1975).
10. A. Aho, R. Sethi, and D. Ullman, 'Pattern matching by parsing', in *Compilers: Principles, Techniques and Tools*, Addison-Wesley, Reading, Mass., pp. 578–579, (1986).
11. G. Reid, 'Efficiency', Usenet group: *comp.lang.postscript*, (No. 4068@adobe.COM), Mountain View, Ca., (1988).