Typographical Extensions for Programming Languages: Breaking out of the ASCII Straitjacket

Paul W. Abrahams
Consulting Computer Scientist
214 River Road
Deerfield Massachusetts 01342
Abrahams@mts.cc.wayne.edu

Abstract

Using extended typography, we can design programming languages that utilize modern display and input technologies, thus breaking out of the ASCII straitjacket. We assume that a language has three representations: a visual representation that describes its displayed form, an internal representation defined for each implementation, and an interchange representation, expressed in pure ASCII, that is defined across all implementations. Using extended typography we can use distinctive typefaces to indicate keywords, thus removing the need to reserve them, and can introduce a variety of new symbols more meaningful than those used in most current programming languages. One benefit is the possibility of arbitrary user-defined operators. We can also introduce new kinds of brackets and methods of pairing brackets visually. Extended typography also helps to solve the problems of writing programs in languages other than English.

Introduction

Programming languages have always been influenced by keyboard and display technologies. The early versions of Fortran used the notation `.GT` to indicate "greater than"; when IBM 026 keypunches were replaced by 029's, Fortran was adjusted to recognize the more natural > symbol. With modern technology we can create a virtually unlimited number of distinctive symbols and display the lexemes of a program in as many different typefaces and sizes as we wish, whether on paper or on a screen. Yet programming language designers have not taken advantage of the opportunities created by these capabilities.

Haunted by the gremlin of syntactic ambiguity, language designers have struggled to make their languages unambiguous yet expressive and readable. They have also struggled to represent a large number of operations using only a small repertoire of symbols. Two assumptions have made these tasks far harder than they need be: (a) that the displayed form of a program must correspond directly to its stored form and (b) that the stored form must be the same for all implementations so that we can transfer programs among implementations. These assumptions are the ASCII straitjacket.

Using extended typography, we can at last break out of the ASCII straitjacket as we design new programming languages. We assume that a language has three representations: a visual representation, an internal representation, and an interchange representation:

- The visual representation, described by a de facto standard, is the form of the language that appears on paper and on display screens. The standard is de facto rather than de jure because of the inevitable small variations in appearance that result from variations in the medium of presentation.

- The internal representation is the one used by a particular implementation to store a program for editing and display. The relationship of the internal representation to the visual one generally need not concern the programmer, and is entirely up to the implementor.
- The interchange representation, which must be standardized across all implementations, is expressed in pure ASCII. Although it should not be utterly inscrutable, it need not be convenient to write. In the interchange language, critical constructs are explicitly labelled, using a notation such as the control-sequence notation of TEX. Thus we can indicate a subset operator by **`\op{subset}`**, which indicates both the syntactic role of the operator and the particular symbol used to represent that operator.

Using the capabilities of modern hardware, we can introduce whatever new symbols we wish and use typographical distinctions to convey syntactic information. Instead of merely typesetting keywords in boldface, we can have our *compilers* recognize boldface as the indicator of a keyword, instantly removing the requirement that keywords be reserved. Moreover, we can easily extend the operators and other lexemes of a language to include a wide range of symbols, just as TEX [Knu84b] does. We can discard the uncomfortable convention that uses := (or even worse, =) as an assignment symbol and employ a more natural symbol such as $\leftarrow$. At last we can use notations such as subscripts and superscripts, either with their mathematical meanings or with other meanings more appropriate for nonnumerical programming.

Background

The idea that hardware character sets need not limit the syntax of a programming language goes back at least to Algol 60 [Nau63]. Algol 60 recognized three language levels: a reference language, a publication language, and a hardware language. The definition of Algol 60 used the reference language, which included operators such as $\supset$ and $\times$ and showed keywords in boldface. (Interestingly, it allowed such operators, as well as the boldfaced logical values **true** and **false,** to appear within strings.) The publication language provided additional flexibility; for example, it permitted subscripts, superscripts, and Greek letters, "according to usage of printing and handwriting". Each hardware representation was to be "a condensation of the input language enforced by the limited number of characters on standard input equipment". Though our visual representation appears at first to be much like the Algol 60 publication language, it actually differs profoundly: while Algol 60 proposed to ignore hardware limitations, we argue that those limitations no longer exist. We also differ from Algol 60 in proposing an interchange language that unlike the Algol 60 hardware language is universally and rigidly defined.

Somewhat the same idea as the Algol 60 language levels showed up in early versions of Lisp [McC60]. John McCarthy, who made major contributions to Algol 60, proposed that Lisp have two forms: M-expressions (meta-expressions) for publication and for ordinary programming, and S-expressions (symbolic expressions) as the hardware representation. However, the S-expression notation (the one with the familiar parenthesis pileups) quickly took over among Lisp programmers because of the lack of computer support for M-expressions.

APL made use of an unusual, though still limited, character set. In [Fal73], Falkoff and Iverson stated that "when one is not concerned with an immediate machine realization of a language, there is no strong reason to so limit the typography and for this reason the language may develop in a freer *publication form*". Nonetheless they soon developed a machine realization, using an APL "golfball" for the IBM Selectric Typewriter with overstriking used to extend the character set beyond the 88 characters available on the golfball. The golfball was not sufficient to ensure wide acceptance of the APL character set, however. During the late 1950's and early 1960's, in a similar spirit, Melvin Klerer at Columbia University explored the use of modified typewriters to make two-dimensional programming languages possible [Kle64].

It is striking that despite advances in both display hardware and keyboards, no one to my knowledge between the late 1960's and today has proposed major extensions to programming language character sets. The designers of Ada, in fact, went to great lengths to ensure that Ada programs could be represented both in ASCII and EBCDIC. To be sure, a great deal of work has gone on in adapting programming languages to natural languages other than English, particularly to the Japanese Kanji character set. This work has had a fundamentally different purpose and nature since it is not aimed at extending the expressiveness and convenience of the programming language itself. Although our approach is not primarily directed at coping with differences among natural languages, it still lends itself to solving that problem.

In "Human Factors and Readability for More Readable Programs" [Bae90], Ron Baecker and Aaron Marcus have explored the possibilities of using radical changes in typography to make C programs more readable and have created a typographical style guide embodying their ideas. The examples of well-presented C programs in their book are extremely impressive—indeed, beautiful. Yet they took the definition of C itself for granted, and only very hesitantly proposed typographical changes that would replace C notations by more readable alternatives. They saw their challenge as program visualization, not as programming language design. Many of their ideas would work even more effectively in a language designed with greater typographical freedom. Indeed, a number of the notations suggested here are also suggested by Baecker and Marcus.

Knuth's WEB language [Knu84a] follows much the same spirit as Baecker and Marcus in attempting to make programs more readable. A WEB presentation can make use of the full typographical facilities of $\TeX$. Yet even WEB does not assume that the underlying programming language uses any kind of extended typography.

Much more recently, the Griffin language under development at New York University uses some non-ASCII operators and states in a preliminary design document that "keywords in Griffin are not reserved words and are distinguished from identifiers in some unspecified way".

Visual, Internal and Interchange Representations

In a programming language that uses extended typography, the visual representation of a program is the representation displayed on a screen or printed on a page. The visual representation is a WSYWIG (what you see is what you get) one in which the appearance of a construct reflects its meaning. WSYWIG has been criticized by Kernighan as meaning "what you see is all you've got", but in this case it may be all you need. Thanks to the capabilities of most display devices, the visual representation presents many opportunities and few problems. On color terminals we can use different colors as well as different typefaces to represent different syntactic elements; the appropriate choices are determined by human factors considerations rather than by technical limitations. Users can customize the presentation according to their individual preferences.

The internal representation of a programming language is the form in which an implementation stores a program for printing and editing. Just as the visual representation of a programming language can be adapted to the needs of readers, the internal representation can be adapted to the needs of editors and output utilities. The internal representation can be different for different implementations; given the variety of ways in which implementations provide different fonts and customized characters, it could hardly be otherwise.

The internal representation of text rarely concerns the user of a WSYWIG editor. Similarly, the internal representation of a program rarely concerns the user of a programmer's editor. Should programmers need to examine that representation directly, we can provide for it with an analogue of the WordPerfect "reveal codes" operation, which shows the internal codes used by WordPerfect and enables the user to edit them.

Although the internal representation is ordinarily very different from the traditional ASCII form of a programming language, the difference is no bar to compilation. Indeed, the internal representation is likely to be easier to compile than the traditional one. Since the internal representation is the primary stored form of a program, conversions among representations will not often be necessary.

It is probably not a good idea to use a parse tree as the internal representation even though the internal representation might well be correlated with a parse tree dynamically updated by an editor. Typographical extensions often appear in contexts such as string literals and constants where the syntactic context does not indicate the typography. Moreover, programs being edited are often in an intermediate state where they are not syntactically correct, a point emphasized by Ballance in the design of the Pan editor [Bal92]. The internal representation therefore cannot depend on a parse tree to determine what the visual representation of a particular construct should be.

The interchange representation is the form used for interchanging programs among implementations. Unlike the internal representation, it uses only a standard printable character set, presumably ASCII. The interchange representation should not be undecipherable, but it need not be easy to peruse. In particular, it can use control sequences such as those of TeX to tag lexical classes explicitly. For example, we can represent an Ω operator using the notation `\op{omega}`. This kind of explicit tagging, which would be inappropriate for a representation intended for people to read, makes it easy to remove potential ambiguities from the interchange representation.

We can compile a program from its interchange representation either by translating it into the internal representation or by compiling it directly. Direct compilation requires an additional front end, in analogy to the way that compilers have different back ends to generate code for different machines and operating environments. Such a front end should not be difficult to write, although since programs are interchanged much less often than they are compiled or examined, the inconvenience of translating programs before compiling them for lack of such a front end is likely to be negligible.

Treatment of Identifiers

Identifiers typically have three uses in programming languages: as structural elements of statements (e.g., **while**), as names of particular infix or prefix operators (e.g., **mod** or **not**), and as names of user-defined or library entities such as variables and procedures. In most languages, identifiers in the first two of these classes are reserved and cannot be used to name variables or procedures. Reserved identifiers have major disadvantages: the programmer needs to remember all of them, and adding new ones to the list can easily break existing programs.

With a great deal of ingenuity, a language designer can avoid the need to reserve structural identifiers. For example, in PL/I one can write such lexical oddities as

```
if if = then then then = else
else else = if
```

However, the grammar needed to make such oddities legal is exceedingly fragile. In any event, even ingenuity cannot avoid the need to reserve the names of infix or prefix operators, given the possibility of sequences such as

```
not not mod mod not mod mod not
```

Using a distinctive typeface for structural and operator identifiers eliminates the need to reserve them and also happens to coincide with the customary way of typesetting programs. We must emphasize the difference here: the usual approach uses the distinctive typeface (normally boldface) as merely a matter of visual presentation, while typographical extensions assign a meaning to the distinctive typeface, a meaning recognized by compilers and editors. In other words, the internal representation of a program contains a marker for each keyword. The editor interprets the marker as meaning "switch to boldface", while the compiler interprets the marker as meaning "this is a keyword".

The PL/I example given above might be written

```
if if = then then
      then ← else
else
      else ← if
```

This example, though thoroughly tasteless, is clear enough.

Typographical extension also creates the possibility of user-defined textual infix and prefix operators. Many modern languages, Ada and C++ for example, provide user-defined operators, but only for a predefined collection of operator names and symbols. Such user-defined operators can be represented by boldfaced identifiers or by special symbols as described below.

Typographical extensions also enable us to enhance the notations we use for user-selected identifiers. For example, we can use subscripts and superscripts in the mathematical style, although we shouldn't take this idea too far; small type is far more difficult to read than large type. We can also introduce Greek letters as well as letters from other alphabets. We discuss below the issue of identifiers written in languages such as Japanese.

Extending the Set of Operators

We can use extended typography to introduce an arbitrary number of new operator symbols and give these symbols their conventional meanings. We can also introduce new symbols for operations unique to computing. For too long, programming languages have been afflicted by odd notations deriving from the choice of graphics in ASCII. Why, for example, should we be using $ and % for notions having nothing to do with currency and ratios? Workers in just about every other scientific field have felt free to create their own notations and to use these notations in their publications. Why should we in the computer field feel constrained not to do likewise? In fact, the history of ASCII is a tale of negotiation among conflicting interests. With a limited supply of characters, users in one field competed with users in another to get "their" characters. The various "national positions" in ASCII, which include square brackets for example, were provided as a way of accommodating these conflicts of interest.

The first, most obvious new symbols are the traditional mathematical operators. Some simple improvements are to replace != by $\neq$, * by $\times$, and ** by $\uparrow$. Other useful mathematical symbols are logical operators such as $\vee$ and $\wedge$ and set-theoretic operators such as $\cap$ and $\cup$. We may also wish to introduce bracketing operators such as the absolute value operator, for example letting $|income|$ denote the absolute value of *income*. This particular operator has some nice generalizations; for example, $|items|$ can denote the length of the list *items*. Paired operators such as this one act as parentheses, an effect that is often useful in itself. Another example of a paired operator is $\lfloor\cdots\rfloor$, which denotes the same thing as **floor**.

In addition, we can introduce symbols for operators specific to computing. The most obvious example is the assignment operator, which we can denote, say, by $\leftarrow$. A natural extension is to allow another operator to be written on top of the assignment operator to denote a modifying assignment, so that $\overset{+}{\leftarrow}$ would denote the same thing as the C operator +=. As another example, we can use the ligature mark $\frown$ to denote the concatenation operator, with $s_1 \frown s_2$ thus denoting the concatenation of s_1 and s_2.

Infix and prefix operators are not the only operator-like symbols that can benefit from extended typography. We can represent conditional expressions using a form such as

$$\begin{array}{ll} cond_1 & \Rightarrow expr_1 \\ cond_2 & \Rightarrow expr_2 \\ & \vdots \\ \square & \Rightarrow expr_k \end{array}$$

with the last line replacing the usual **else** part. We can use additional kinds of brackets beyond the standard three (parentheses, square brackets, curly braces) to represent additional constructs. We can use different sizes of parentheses to indicate nested groupings, just as mathematicians do, and even enhance such a notation to provide for labelled parentheses. We can also internally associate an operator with its parentheses, a device that can help a programmer avoid mismatched parentheses.

Comments and String Literals

Typographical extensions enable us to include arbitrary text in comments. For example, we can typeset a program fragment within a comment as we would within the program proper. A prettyprinter can't do that since it doesn't have the necessary syntactic context. Graphics within comments are more problematic, however, because there is no obvious way to represent them in the interchange language. Standardizing the graphics representation seems far beyond the scope of designing a programming language.

String literals, unlike comments, are used as data by programs. It is therefore unclear how far we can go in allowing typographical extensions within them. We certainly can use typographical extensions to represent control characters more conveniently than we usually do. For example, we can indicate a tab with a special symbol such as ▶ or with an indicator such as $\boxed{\text{TAB}}$. The specifics of what is permitted in a string literal and how a literal is interpreted are best left to the designer of a particular programming language.

Treatment of Foreign Languages

The fact that programming language are almost always written in English has long been a source of aggravation for those having a different native language. Programming language keywords are English-specific and the structure of programming language statements is sometimes affected by English word order. For users of languages such as Japanese or even Russian that have different character sets, the role of English has been particularly troublesome. Program listings, for example, are often written out by hand so that the comments can be in the programmer's native language.

Typographical extensions make it possible to store programs in a form that is as natural to Japanese, say, as the traditional form is to English. Keywords, identifiers, and comments can all be written using Japanese characters. (I don't presume to address here the problem of providing Japanese input, which affects all kinds of computer applications.) In fact, we can even compile a program written in Japanese within an implementation that directly supports only English.

The critical issue in supporting foreign languages is the relation between the internal representation and the interchange language—or looking at it another way, how the interchange language supports keywords written in a foreign language and characters selected from an arbitrary character set. For keywords, the solution is straightforward: the interchange representation is always in English. Chauvinistic as that may sound, it imposes no real burden on Japanese programmers, say, since they are nearly always working with the visual representation (and indirectly with its reflection in the internal representation). For identifiers, comments, and other textual objects whose contents are not fixed by the programming language, we can use an escape notation, similar to the one we use for operators, to insert non-ASCII characters into text. The interchange representation should, of course, be standardized among *Japanese* implementations, but it is of no concern to English implementations. In general, an implementation can provide an arbitrary visual representation for characters it is not familiar with; as far as the internal representation goes, its only concern is to preserve the semantics of these characters.

In preserving semantics, there are three cases: comments, identifiers, and string literals. For comments, there are no semantics to preserve. For identifiers, the only semantics that need to be preserved are the equivalence of identifiers with the same spelling. String literals pose a more thorny problem, but not one peculiar to extended typography. Since an English-only implementation cannot print or display these literals in their native form, the most we can expect is to preserve the semantics of operations such as concatenation, length, and substring extraction. The treatment of large character sets in C provides a promising model of how to deal with these issues; see Plauger's discussion of that treatment [Pla92].

Editing Considerations

Languages that use typographical extensions require editors different from the ones that most programmers are accustomed to. Such creaky relics as the `vi` text editor will not work with these languages. An appropriate editor is more like one of the WSYWIG editors such as WordPerfect, Word for Windows, or Ami Pro that are so widely used on personal computers. Nevertheless, even the UNIX world has recognized the new world of text editors. The FrameMaker editor, a commercial product, has become quite popular among UNIX users (and has a compatible implementation under Microsoft Windows); two other examples of modern editors in the UNIX world (though still research projects) are the Lilac editor developed by Brooks at the DEC Systems Research Center [Bro91] and the Pan editor from UC Berkeley [Bal92].

Though typographical extensions are harder to type, this is not a major disadvantage for two reasons. First, most WSYWIG editors already provide methods of controlling typography and producing characters not appearing directly on the keyboard. Control and meta shifts provide simple ways of introducing font changes. In a graphical environment, a set of symbols can be displayed on a screen with one being selected using either the keyboard or a mouse. Ami Pro uses this method for selecting mathematical symbols to appear in displayed equations. Second, we have come to recognize that most of the time spent working with a program is spent in reading it, not in entering it into the computer. The extra effort required to type a program is small compared with the total effort required to get that program working.

Conclusion

Typographical extension may seem like such an obvious idea that you might wonder why it has not already been widely adopted. It applies equally well to specialized languages and to general-purpose ones, to functional languages and to imperative ones. The critical advance I propose in this article is the distinction between the internal representation and the interchange representation. Without such a distinction, the representation becomes overburdened and cannot meet the simultaneous constraints of readability, unambiguity, and commonality among implementations.

The limitation on extending syntax through typography is not what our computers can do; it is what our eyes and minds are capable of perceiving. A language designer applying the idea of extended typography must allocate meanings among typefaces and symbols carefully—beyond a certain point, more is less. The syntactic conventions of the Algol 68 report [vWi75] provide a striking example of how an excess of typographical distinctions can leave the reader utterly confused. Used with discretion, however, typographical freedom can bring a new level of expressiveness to programming languages and their users.

References

[Bae90] Baecker, Ronald, and Marcus, Aaron, Human Factors and Typography for More Readable Programs: ACM Press/Addison-Wesley, Reading Mass., 1990.

[Bal92] Ballance, Robert A.; Graham, Susan L.; and van de Vanter, Michael L., The Pan language-based editing system: *ACM Trans. Softw. Engg. and Methodology* **1**(1), Jan 1992, p. 95–127.

[Bro91] Brooks, Kenneth, Lilac: a two-view document editor: *IEEE Computer* **24**(6), Jun 1991, p. 7–19.

[Fal73] Falkoff, A.D. and Iverson, K.E., The design of APL: *IBM Jnl. of Res. & Dev.*, Jul 1973, p. 324–334.

[Kle64] Klerer, M. and May, J., An experiment in a user-oriented computer system, *Comm. ACM* 7(5), May 1964.

lKnu84a] Knuth, Donald E., Literate Programming: *The Computer Jnl.* **27**(2), p. 97–111.

[Knu84b] Knuth, Donald E., The TEXbook: Addison-Wesley, Reading Mass., 1984.

[McC60] McCarthy, John, Recursive functions of symbolic expressions: *Comm. ACM* **3**(4), Apr 1960, p. 184–195.

[Pla92] Plauger, P.J., Large character sets for C: *Dr. Dobb's Jnl.* #191, Aug 1992, p. 16–24.

[Nau63] Naur, Peter, ed., Report on the algorithmic language Algol 60: *Comm. ACM* **6**(1), Jan 1963, p. 1–17.

[vWi75] van Wijngaarden, A.; Mailloux, B.J.; Peck, J.E.L.; Koster, C.H.A.; Sintzoff, M.; Lindsey, C.H.; Meertens, L.G.L.T.; and Fisker, R.G., Revised report on the algorithmic language Algol 68: *Acta Informatica* **5**, 1975, p. 1–236.