ACM *Ada Letters* **XIII**,5 (Sept./Oct. 1993), 43-47.

Strategies for the Lossless Encoding of Strings as Ada Identifiers

Henry G. Baker

Nimble Computer Corp., 16231 Meadow Ridge Way, Encino, CA 91436 (818) 986-1436 (818) 986-1360 (FAX)

When translating software from other languages into Ada, it is necessary to translate foreign identifiers into Ada names. If the foreign language provides a richer syntax for identifiers than Ada—e.g., it allows non-alphanumeric characters—then some encoding scheme will be required in order to provide a readable 1-1 mapping. Schemes for achieving such 1-1 mappings are discussed.

Mappings of this sort could be useful in automatic translators which convert C/C++ header "`.h`" files into Ada, as well as for translators of high-level design and/or specification languages into Ada.

1. Introduction

During the translation of programs written in other languages into Ada, it becomes necessary to translate foreign identifiers into Ada identifiers. For a small program, an *ad hoc* approach will work, but for larger programs some automatic method must be used, or else the translator will run the risk of inadvertent name collisions. An automatic method has the additional advantage that the translator need no longer be imaginative, and translation productivity will therefore be enhanced.

If the translation into Ada is being performed by an automatic tool, it is essential that its algorithm for identifier translations produce names which do not collide with each other, with Ada reserved words, or with usual Ada names.

Although there have been several papers about translators of other languages into Ada [Albrecht80] [Wallis85] [Kaelbling86a,b], there has been little discussion of the particular problem of identifier translation. One must presume that the users of such translators were willing to put up with incomprehensible names, or were able to utilize relatively simple *ad hoc* schemes—e.g., Fortran-77 identifiers, which consist of 1-6 uppercase alphanumeric characters starting with an uppercase letter, map trivially[1] into Ada names [ANSI-Fortran].

Nimble Computer Corporation has developed an automatic translator from the Common Lisp programming language into Ada [Baker89], and hence has faced the problem of mechanically translating Common Lisp identifiers—called "symbols"—into Ada. As Common Lisp symbols have spellings which consist of completely arbitrary character strings, it became necessary to devise a 1-1 mapping from ASCII strings into legal Ada identifiers. However, although a hexadecimal mapping of "`Hello`" into `X_48_65_6C_6C_6F` is simple and straightforward, it is also completely unreadable, and thus makes for a poor quality translation of Common Lisp into Ada. We discuss mappings which are both 1-1, and also readable.

2. Ada Identifiers

Ada identifiers are used both as names and as reserved words [AdaLRM, 2.3/1, 2.9/2]. Ada identifiers have the following syntax: `letter {[underline] letter_or_digit}` [AdaLRM, 2.3/2]. In other words, Ada names: 1) must be comprised of letters, digits and underlines; 2) must start with a letter; 3) must not end with an underline; 4) must not have adjacent underlines; and 5) must not be one of the 63 reserved words.[2] Furthermore, Ada identifiers cannot be longer than one line, because "the end of line is always a separator" [AdaLRM, 2.2/3]; e.g., we are familiar with one validated Ada compiler which has a line length limitation (and hence an identifier length limitation) of 200 characters. Finally, Ada does not distinguish the alphabetic case of letters—i.e., the case of identifiers is usually normalized to upper case in Ada formatters ("pretty printers") and in the `'IMAGE` attribute of enumerated types (Ada does, however, preserve the case of individually quoted enumerated characters—e.g., `'a'`)[3].

3. Desirable Characteristics of String-to-Identifier Mappings

Below, we list a number of desirable characteristics of any string-to-identifier mappings:

1. The *range* of any mapping must be the set of legal Ada identifiers—i.e., only letters, digits and underlines, first character a letter, last character a letter or digit, and no adjacent underlines.

---

[1]Except for Ada reserved words.

[2]A few more reserved words are on the way with Ada-9X [Ploedereder92].

[3]Thanks to Geoff Mendal for pointing this out, as well as for his other excellent suggestions.

2. No string may map into any Ada reserved word—e.g., no string should map into the Ada reserved word **`ACCEPT`**, etc.

3. The mapping must be 1-1—i.e., for no two strings $s_1 \neq s_2$, should $m(s_1)=m(s_2)$, for any mapping m. In applying this restriction, we understand that the "=" compares identifiers the same way that an Ada compiler does—by ignoring alphabetic case.

4. The mapping should be approximately homomorphic with respect to length—i.e., if $l(s_1) \approx l(s_2)$, then $l(m(s_1)) \approx l(m(s_2))$. In other words, very short strings should not map to very long identifiers, nor should very long strings map to very short identifiers.

5. The mapping should be straightforwardly and locally computable in both directions—i.e., the inverse mapping m^{-1}, which exists because m is 1-1, should be easy to compute.

6. The mapping should be readable and intuitive, so that it may be computed by inspection.

4. Morphology of Lisp Symbol Names

Since our primary purpose in translating strings into Ada names is translating Common Lisp [Steele90] symbol names, we review the typical structure of Common Lisp symbol names.

Although any Common Lisp string may be used to name a Common Lisp symbol, the most common symbols have alphanumeric names, which look very similar to the identifiers used by other programming languages. In particular, most standard Common Lisp symbols consist of alphanumeric characters which are punctuated with a hyphen, rather than the underline beloved by Ada and C. Thus, although underline is a legitimate character in a Lisp symbol, the hyphen is traditionally preferred by Lisp programmers for intra-symbol delimitation. Examples of standard Lisp symbols are `array-element-type` and `double-float-negative-epsilon`.

Common Lisp symbols may also contain other non-alphabetic characters, however. For example, the usual arithmetic/logical operators `+`, `-`, `*`, `/`, `=`, `<=`, `/=`, etc., are all normal Lisp symbols. Furthermore, non-alphanumeric characters and normal characters may be intermixed—e.g., `char<`, `string>=`, `&rest`. In particular, most variables with global scope are given names with surrounding asterisks—e.g., `*standard-output*`.

A naming convention that Common Lisp shares with older languages like Fortran and Ada is case-insensitivity. For example, the spelling of the Lisp symbol `min` is the string `"MIN"`—i.e., the spelling is normalized to upper case. The symbol whose spelling is `"min"` can be typed in as `\m\i\n` or `|min|`,[4] but the unadorned `min` means the upper case spelling "`MIN`".

Due to this preferred usage, the most readable mappings of Lisp symbols to Ada names will substitute underlines for hyphens, and will preserve the alphanumeric subsequences within the spelling of the symbol.

5. Developing a Readable Mapping

We will develop a readable mapping from strings into Ada names by starting from a simple mapping, and then elaborate it until it can handle the complete range of strings.

The most obvious mapping of uppercase alphanumeric strings with an initial letter is the identity mapping—e.g., `plus` becomes `PLUS`. This mapping is short, computable and obvious, but it cannot cope with the wide variety of actual strings, nor can it be extended in any way. Furthermore, it makes no provision for avoiding Ada reserved words, many of which are commonly used Lisp symbols.

A more extensible mapping scheme incorporates a standard prefix and/or suffix to distinguish mapped symbols from Ada reserved words. A typical prefix might be "S_", meaning "Symbol", followed by the mapped characters themselves. This mapping scheme maps the symbol `accept` into the Ada identifier `S_ACCEPT`, which is not the same as the identifier `ACCEPT`, and therefore avoids collisions with reserved words. Although the mapping of non-reserved words like `plus` into `S_PLUS` instead of the more elegant `PLUS` is unfortunate, this prefix scheme has the advantage of uniformity and isolation from additions to Ada's set of reserved words.

The prefix and/or suffix schemes also have the advantage of allowing for extensions over time to the basic mapping scheme—e.g., different prefixes can represent mappings for different languages.

So far, we can map only uppercase alphanumeric symbols into Ada identifiers without fear of collisions with Ada reserved words. We still cannot handle the hyphenated words typical of Lisp symbols, nor can we handle non-alphanumeric characters like "<" or "=".

---

[4]Backslash "\" quotes one character; vertical bars "|" are used in pairs (analogous to ") to quote character sequences.

We could make the straight-forward replacement of the hyphen "-" by the underline "_", but such a replacement would lose the distinction between the symbols abc-def and abc_def, and make an illegal Ada identifier from the symbol abc--def. We therefore require a more sophisticated replacement rule.

We consider the hyphen "-" in Lisp symbols and the underline "_" in Ada identifiers to separate explicitly delimited *syllables* from one another. We will therefore use the explicit hyphen "-" to parse Lisp symbol names into a list of non-empty syllables which will then be translated more-or-less separately and then concatenated together with the Ada syllable underline delimiter "_". If we ignore, for the moment, the problem of adjacent hyphens and final hyphens, then we achieve the reasonable mapping of the symbol output-file into S_OUTPUT_FILE and the symbol find-if-not into S_FIND_IF_NOT.

We must now deal with the problem of symbols like char<=. This symbol is usually pronounced as "char less than or equal" or "char less equal", so if we take our cue from the symbol's pronunciation, we should map this symbol into something like S_CHAR_LESS_EQUAL. Such a mapping requires that each non-alphanumeric character be considered its own syllable, which is translated independently of the other syllables. If we extend this idea, then we must define names for all of the non-alphanumeric characters, as the following table illustrates. However, to minimize collisions with user-defined syllables, the names for the non-alphanumeric characters should *not* be real English words, but some contracted form of a name for the character—e.g., DLLR rather than DOLLAR. The use of contraction allows the reader to understand the syllable, but also reminds him that the syllable is not a literal syllable, but represents a single character.[5]

```
!     BNG               -- bang.  "exclamation point" is too long!
"     QUT2              -- "double quote" is too long
#     SHRP              -- sharp.  "pound sign" is too parochial
$     DLLR              -- dollar.
%     PCT               -- percent.
&     AMPR              -- ampersand.  "and" assumes C usage.
'     QUT1              -- "single quote" is too long
(     LPAR              -- "left parenthesis" is too long
)     RPAR              -- "right parenthesis" is too long
*     STR               -- star.  "asterisk" is too long
,     CMMA              -- comma.
:     CLN               -- colon.
;     SMICLN            -- semicolon.
<     LSS               -- less.  "less than" is too long
=     EQUL              -- equal.  "equal to" is too long
>     GRT               -- greater.  "greater than" is too long
?     QST               -- question.  "question mark" is too long
@     ATSGN             -- at sign.  "at" is to ambiguous
[     LBRK              -- "left square bracket" is too long
\     BSLSH             -- "back slash" is too long
]     RBRK              -- "right square bracket is too long
^     UPARW             -- up arrow.  "circumflex" is to pedantic
_     UNDR              -- under.  "underline" and "underscore" are too long
`     BQUOT             -- "back quote" is too long
{     LBRC              -- "left curly brace" is too long
|     VBAR              -- "vertical bar" is too long
}     RBRC              -- "right curly brace" is too long
~     TLD               -- tilde.
+     PLS               -- plus.
.     PRD               -- "period" is too long
/     SLSH              -- slash.  "divided" ignores "not" (/=) connotation
... continue with definitions for other non-alphanumeric characters.
```

Using this table, we can now map mixed symbols like string/= into S_STRING_SLSH_EQUL. Unfortunately, this symbol now collides with the mapped symbol string-slsh-equl, so we must somehow distinguish the

[5]Thanks to Jorge Diaz-Herrara for pointing out the names in Ada's ASCII package [AdaLRM,C.], as well as a bug. Unfortunately, Ada's ASCII character names are unsuitable due to their length and potential for collisions.

ACM *Ada Letters* **XIII**,5 (Sept./Oct. 1993), 43-47.

two. Since `string/=` occurs more frequently than `string-slsh-equl` (probably because `string/=` is usually pronounced "string not equal"), we would like to map `string/=` into `S_STRING_SLSH_EQUL`, and then choose a longer mapping for the symbol `string-slash-equal`.

In the best of all possible worlds, we would utilize doubled underlines "__" to distinguish alphanumeric syllables which actually occur from those which are mapped from characters. Unfortunately, these double underlines are illegal in Ada (sigh!). We must therefore choose a more devious scheme which is consistent with the English language. Since the letter "q" is rarely used in English at all, and then even more rarely without the letter "u" succeeding, we will indicate actually occurring syllables which collide with our table entries with a prefix of "`QQ_`", for "quote". In other words, we will map the symbol `string-slsh-equl` into `S_STRING_QQ_SLSH_QQ_EQUL`. Since the syllable after the "`QQ_`" is treated literally, any syllable which actually is "`qq`" is simply prefixed with another "`QQ_`" so the mapping is 1-1. For example, the symbol `string-qq=` is mapped into `S_STRING_QQ_QQ_EQUL`.

Now that we have the "`QQ`" mechanism for quoting syllables, we can eliminate the initial "`S_`" prefix from most symbols, and more uniformly utilize "`QQ_`" prefixes on those identifiers whose first syllable requires quoting—e.g., any Ada reserved word; `accept` thus becomes `QQ_ACCEPT`. In the unlikely event that the Lisp code uses the symbol `qq-accept`, it will map to `QQ_QQ_ACCEPT`.

We now take up the problem we deferred above—i.e., the problem of doubled and final hyphens. This problem is easily solved by treating all hyphens in a multiple-hyphen sequence, except for the first one, as separate single-character syllables, to be encoded separately according to the table which now requires an additional entry: "`-`" is mapped to "`MNS`". Furthermore, a final hyphen is also treated as a separate one-character syllable. Thus, the single-hyphen symbol "`-`" is mapped into `MNS`, the two-hyphen symbol "`--`" is mapped into `MNS_MNS`, the three-hyphen symbol "`---`" is mapped into `MNS_MNS_MNS`, the symbol `abc-def` is mapped into `ABC_DEF`, the symbol `abc--def` is mapped into `ABC_MNS_DEF`, and the symbol `abc-def-` is mapped into `ABC_DEF_MNS`.

The final problem to be solved is that of lower case symbols like `\f\o\o` (or `|foo|`). Actually, we have several problems—the problem of whole lower-case syllables, the problem of a syllable with an initial uppercase letter followed by lower case alphanumerics, and the problem of isolated lower-case letters. Common Lisp itself provides for an isolated lower case letter via the single-character quoting mechanism of "\", and retains the ability to quote an entire string using vertical bars, as above. Unfortunately, we must both quote and translate, since we cannot represent non-alphanumeric characters. Extending the mechanism suggested above, we can map an entire lower-case syllable through a "`QQL_`" (quote lower-case) prefix, or map a syllable with an initial upper-case letter through a "`QQD_`" (quote 1 character, then down-case) prefix. Finally, the case of the single isolated lower-case character can be handled by treating it as a separate syllable, and providing it with a default mapping in the character mapping table—e.g., the single lower-case letter "`a`" could be mapped to a separate syllable "`LCA`".

6. Examples and Discussion

We have shown a reasonably readable encoding method for translating Lisp symbol names into legal Ada names which is both 1-1 and readable. Here are some examples:

```
i,n                           I,N
character, integer            CHARACTER, INTEGER
array, null                   QQ_ARRAY, QQ_NULL            (Ada reserved words)
let, let*                     LET, LET_STR
*standard-output*             STR_STANDARD_OUTPUT_STR
&allow-other-keys             AMPR_ALLOW_OTHER_KEYS
*, **                         STR, STR_STR
-, --                         MNS, MNS_MNS
1+, 1-                        QQ_1_PLS, QQ_1_MNS     (1st char. non-alphabetic)
<, >                          LSS, GRT
=, /=, <=, >=                 EQUL, SLSH_EQUL, LSS_EQUL, GRT_EQUL
char<, char<=                 CHAR_LSS, CHAR_LSS_EQUL
|hello|, |Hello|, hel\lo      QQL_HELLO, QQD_HELLO, HEL_LCL_O
()                            LPAR_RPAR             (a spelling for Lisp's nil!)
```

Our scheme handles the 700-odd standard[6] symbols of Common Lisp, and represents most user symbols in a moderately readable fashion. Due to the implicit length limitation of Ada symbols, some excessively long Lisp

[6]There are no "reserved" symbols in Common Lisp; but it is difficult to use about 30 symbols as function names.

symbols may not be mappable, but since most long Lisp symbols are hyphenated English phrases, these will usually map into Ada names of the same length.

Some of these ideas should also be useful in the conversion of identifiers from non-Lisp languages into Ada, since the identifiers of these languages are more constrained, and hence easier to map into Ada. In particular, Ada programs which must interface with C and C++ programs—e.g., MIT's X-Windows and Microsoft Windows—have found the need to automatically translate C/C++ header "`.h`" files into Ada, and such translators could utilize some of the strategies suggested here.

Finally, these techniques may also be useful for the automatic mapping of other kinds of object names into Ada—e.g., in CAD/CASE tools, such as high level "specification" languages which incorporate Ada code generators.

7. References

[AdaLRM] *Reference Manual for the Ada® Programming Language*. ANSI/MIL-STD-1815A-1983, U.S. Gov't Printing Office, Wash., DC, 1983.

[Albrecht80] Albrecht, P.F., *et al*. "Source-to-source translation: Ada to Pascal and Pascal to Ada". *Proc. ACM Sigplan Symp. on Ada, Sigplan Not. 15*, 12 (1980), 183-193.

[ANSI-Fortran] American National Standard Programming Language FORTRAN. ANSI X3.9-1978, NY, NY, 1978.

[Baker89] Baker, H. "The NIMBLE Project—Real-Time Common Lisp for Embedded Expert System Applications". *Proc. 1989 AIAA Computers in Aerospace Conf.*, Monterey, CA, 1989.

[Kaelbling86a] Kaelbling, L.P. "FORADA: A FORTRAN to Ada Translator". *Ada Letters VI*, 6 (1986), 107-108.

[Kaelbling86b] Kaelbling, L.P. "The Role of Translators in an Ada Environment". *Ada Letters VI*, 6 (1986), 105-106.

[Martin86] Martin, D.G. "Non-Ada to Ada Conversion". *Ada Letters VI*, 1 (1986).

[Parsian88] Parsian, M., *et al*. "Ada Translation Tools Development: Automatic Translation of FORTRAN to Ada". *Ada Letters VIII*, 6 (Nov.-Dec. 1988), 57-71.

[Ploedereder92] Ploedereder, E. "How to Program in Ada9X, Using Ada83". *Ada Letters XII*, 6 (Nov/Dec 1992), 50-58.

[Steele90] Steele, G.L. *Common Lisp: the Language, 2nd Ed*. Digital Press, Bedford, MA, 1990.

[Wallis85] Wallis, P.J.L. "Automatic Language Conversion and Its Place in the Transition to Ada". *Proc. Ada Int'l. Conf. "Ada in Use",* Barnes & Fisher, *eds*., Camb. Univ. Press, 1985, pp.275-284.

