Porting *Pan I* to Allegro COMMON LISP*

Darrin J. Lane

September 6, 1988

Abstract

This document describes the process of porting *Pan I* from Franz Lisp to Allegro COMMON LISP. It focuses on issues relevant to *Pan*, but is intended to supply guidelines on porting any large Lisp system. The steps taken to translate the code were designed to keep the Franz implementation functional while the Allegro implementation was being developed. Three techniques were used. Franz code was rewritten to be compatible with both Lisps. Franz macros were introduced to duplicate Allegro functionality. Lastly, conditional compilation was used to allow the coexistence of both Franz and Allegro versions of code in situations that proved too cumbersome for the first two techniques.

1 Introduction

Pan I is a multilingual language-based editing and browsing system developed at the University of California, Berkeley[2],[3]. The system was originally implemented in Franz Lisp[4]. This document presents the process by which it was ported to Allegro COMMON LISP[1]. The decision to port was based on the need for greater efficiency with the forthcoming addition of semantic analysis[1]. It was also felt that COMMON LISP would give us a wider user community and greater portability. Throughout this paper Franz will refer to Franz Lisp and Allegro to Allegro COMMON LISP.

Early during the planning stages of the port it was decided that the Franz version of the code would be kept in working order as long as possible.

*Sponsored by the Defense Advanced Research Projects Agency (DoD), monitored by Space and Naval Warfare Systems Command under Contracts N00039-84-C-0089 and N00039-88-C-0292 and by the Bell Communications Research Full-Time Graduate Study Program .

[1]Robert Ballance is currently working on *Pan*'s semantic analysis phase.

At that point we envisioned making some of the compatibility changes to the existing system, but would then "throw the switch" to Allegro leaving Franz behind forever. Unfortunately, this would have also left us without a functional *Pan* for an indeterminate amount of time. We were unhappy with this prospect, but did not see any alternative. Later, as the project progressed it became evident that through the use of a few simple techniques we could port to COMMON LISP and still maintain an efficiently running Franz version. In fact, the same source code can now be used to generate both Franz and Allegro versions of *Pan*.

Three techniques were used to port the system without allowing it to be nonfunctional for long periods of time: (1) code was rewritten to be compatible with both Lisps, (2) complex macros were used to extend Franz to include large amounts of Allegro functionality and (3) conditional compilation allowed the coexistence of both Franz and Allegro versions of code. Although, changes to the system can most easily be classified by the primary technique involved, many required a combination of methods to achieve the desired result. Secondary techniques are noted accordingly throughout the report.

This document is organized in the following manner. Section two explains what preparation was necessary before the port. The third, fourth and fifth sections discuss the translation techniques given above. Sections six through eight present the major translations necessary. Section nine discusses enhancements made to the system during the port. The tenth section explains functionality lost in the port. Section eleven summarizes some of the outcomes of the port. Finally, section twelve provides some concluding remarks.

It is assumed that readers of this document are familiar with Franz Lisp and Allegro COMMON LISP and have read both *Pan I: An Introduction for Users*[2] and *The Architecture of Pan I*[3].

2 Getting Started

The first two months of the port were dedicated almost entirely to familiarizing myself with *Pan* and the rest of my working environment. Since I was not the author of the system, there were many things I needed to learn before I could begin translating the code. Bringing myself up to date on a project that had been evolving for upwards of three years was quite a challenge. First, I read the COMMON LISP manual [5] in order to gain

exposure to the functions available in the language. I had programmed in Lisp, but I was not experienced with Lisp systems programming. A working knowledge of nearly every function provided by both Lisps was required and reading the manual seemed like a step in that direction. Next, I read the *Pan* documentation, experimented with the system and explored the source files in order to learn about *Pan* itself. From these efforts I gained familiarity with the coding style and some perspective on the user's view of *Pan*. During this time I also met with other members of our research group to ask questions and to discuss issues relevant to the port. These meetings began to provide me with a "feel" for the code and the system as a whole that would have taken a great deal of time to acquire on my own. By the end of this adjustment period I had developed a list of porting issues and had begun making the first changes to *Pan*.

3 Franz Code Rewrites

3.1 Cleanup

The first series of Franz rewrites were intended to reorganize the system in preparation for the port. Code was rewritten to remove unnecessary functionality, reduce complexity where possible and standardize conventions.

While examining the source code it was discovered that the fully general functionality provided by two modules was not utilized. In the first instance, a module devoted entirely to the construction and maniplation of variable length strings had been developed. Variable length storage of strings is necessary when transforming user selections to strings and copying edit buffer text lines for regular expression searching. In these situations a buffer of sufficient size is allocated once and reused by overwriting it and marking the end. To eliminate the need for this "vstring" module, conditional compilation was used to implement the storage with fill pointer arrays in Allegro. The Franz system was then modified to fake the fill pointer capability by using a null character to mark the end of the string within a standard array. Finally, the "vstring" code was removed from the system to avoid unnecessary effort in porting. Another overly general and under-used facility allowed the user to provide the system with a string that would specify the format for a status line. A set of functions had been written to interpret the format string and alter the display of the status line accordingly. This was done elegantly, but the status line was very rarely used. In addition, with the advent of SunView's panel display, this functionality was obsolete. Most

of the information available through the status line was now displayed in the panel of the viewport window. A command for displaying a statically formatted string of the remaining information was written and the rest of the code was removed.

Although *Pan* had been written primarily by one person, many coding styles could be found in the implementation. This was due to the fact that Franz Lisp itself had changed quite dramatically during the three years of *Pan* evolution. For example, large portions of the system existed before Franz had a packaging capability. Early naming conventions showed an attempt to divide up the name space by prepending functions with the module name and an exclamation point. The final phase of the cleanup (which continued throughout the entire port) was directed towards updating and standardizing conventions such as these, in addition to improving the general appearance of *Pan*'s code. Files, functions and variables were renamed to more accurately reflect their purpose. Comments were added as the functionality of code was understood. Lastly, the require and export lists of each module were updated to correctly reflect the state of the system.

3.2 Compatibility Changes

As mentioned earlier, compatibility with Allegro was increased by rewriting Franz code to be compatible with both Lisps. This phase provided the ideal environment for "getting my feet wet." At first, small, isolated pieces of code were modified, but later as my knowledge of *Pan* grew, larger sections were rewritten. Starting off slowly meant being able to recover from mistakes easily, which was essential for experimenting with and learning about the system.

The following is a list of the compatibility changes:

- Explicit references to the ports `poport`, `piport` and `errport` were removed, since the functions `format`, `read` and `warn` use them as the default.
- Uses of the functions `tyo`, `tab`, `terpri` and `msg` were replaced exclusively with `format`.
- Compiler directives which toggle macro compilation and declare specials were removed whenever possible. Those that remained were incorporated into the Franz extension package.

- Since we had chosen to use a case insensitive version of Allegro, many functions were renamed to avoid clashes.
- "Setting" functions such as `addhash` and `vset` were changed to `setf`'s on the accessor functions.
- Uses of the macro `loop` were changed to `do` and `do*`. Although a `loop`[2] macro is also available for Allegro, it was felt that the "do" iteration styles were more appropriate within a Lisp system.
- The function `alphalessp` was replaced with `string<` where it was used on strings.
- Since Allegro offers no fixnum specific operations, the fixnum operators `<&`, `>&`, `<=&`, `>=&`, `=&`, `minus`, `plus` and `times` were replaced with the generic `<`, `>`, `<=`, `>=`, `=`, `-`, `+` and `*`, respectively. Efficiency was lost with this change, however, so Franz macros were used to alias the fixnum specific operations by their generic counterparts. This was possible since *Pan* uses fixnums for all its mathematical calculations. The Allegro version of *Pan* will have to wait to regain the lost efficiency until appropriate compiler switches are made available.
- The function `concat` was replaced by an `intern` applied to `format`. For example, forms such as `(concat 'sym i)` would be changed to `(intern (format nil "sym~D" i))`.
- Uses of the function `if` with the keywords `then` and `else` were rewritten with `when`, `unless`, `if`, `cond` and `case`. The Allegro function `excl:if*` was added to the extension package and used in some cases as well.

3.2.1 Defstructs

The code for defining and manipulating structures required several changes to be compatible with both Franz and Allegro. For that reason it is explained in more detail here. First, the `defstruct` keyword argument `:conc-name` was used to ensure that all structure accessors would be concatenated with the structure name. Uses of the accessors were then changed to contain the concatenation. In our Franz system the accessor names had not been

[2]The `loop` macro referred to here has the keywords `for`, `then`, `return`, etc. It is not the standard COMMON LISP `loop` which generates an infinite loop.

concatenated consistently, since no specification of the `:conc-name` argument defaults to no concatenation. However, in Allegro the default is to use the name of the structure. By explicitly requiring the concatenation both Lisps exhibit the same behavior. In the future `:conc-name` references can be removed from the Allegro system. Next, calls to the "make" functions created by `defstructs` were changed to use the slot names only as keyword arguments. Franz accepts either symbols or keywords, but Allegro insists upon the latter. Finally, since Allegro does not associate "alter" functions with structures, their uses were changed to `setfs` on the slot accessors.

After the above changes were completed the structure manipulations were compatible in both Lisps; however efficiency considerations required one final alteration. Creation of the data structures necessary for managing text happens frequently within inner loops of *Pan* and therefore cannot afford the time it takes to match up keyword arguments. To eliminate this overhead `:constructor` was used to declare a constructor function with only required arguments. The most commonly initialized slots were set by the call to the constructor and the remaining ones were initialized using `setfs` on the accessor functions.

4 Macro Introduction

In order to keep the Franz version of the system functional during the port, macros were introduced into our extension package[3] to increase compatibility with Allegro. These macros defined Allegro constructs in terms of those that exist in Franz. This technique had several advantages. First, macros allowed the Franz system to remain efficient. Second, by introducing Allegro constructs into a running system they could be tested and debugged immediately. Finally, code duplication was kept to a minimum, since creating Franz and Allegro versions of code was unnecessary.

This method employed three distinct types of macros. What follows is an explanation of their differences along with some examples to provide insight into how the macros were used in practice.

The first type of macro simply equates the function or macro definition of two symbols. In this way Allegro names could be attached to Franz constructs. For example, the form `(defnewname consp dtpr)`[4] defines the

[3] Many of the macros in *Pan*'s extension package were written by Jacob Butcher.

[4] `defnewname` is also a macro in the extension package. It defines the first argument in terms of the second.

Allegro name consp as an alias for dtpr. Notice that the two functions have the same definition in both Lisps. However, renaming need not be confined strictly to functions having equivalent behavior. Similar behavior will suffice if the usage of the newly defined name is limited to cases covered by the original function. For example, **aref** can be defined in terms of **vref** if it is used only on vectors. Section 1 of Appendix A contains a table listing the aliases defined with **defnewname**.

In situations where more than renaming was needed, Allegro constructs were defined by cleverly combining existing Franz functionality. For example the following macro was used to define the COMMON LISP function **merge**.

```
(defmacro merge (result-type seq1 seq2 comp-pred)
  '(insert (car ,seq1) ,seq2 ,comp-pred t))
```

Notice that this definition does not support the :key keyword argument. Furthermore, it will operate only on lists, not general sequences. As with renaming, an exact match between the Franz and Allegro functionality is unnecessary. The uses of the function dictate the level at which the equivalence can be assumed. A large amount of effort was saved by implementing only functionality actually used by the Franz system. Section 2 of Appendix A contains the macro definitions from our Franz extension package used in this technique.

The final class of macros used in the port provided Franz functionality to Allegro. This technique was applied only to the functions **memq**, **remq**, **rassq**, **assq** and **delq**. These deserve special treatment for several reasons. First, the functions are easily written in Allegro using **member**, **remove**, **rassoc**, **assoc** and **delete**, respectively with a **:test** argument. Second the Allegro functions are not easily written in Franz. Lastly, the macros can be removed and replaced with their definitions when all ties with Franz are broken. As a general rule however, this is an inappropriate technique for porting Franz code to Allegro and is therefore not recommended.

5 Conditional Compilation

During the port, keeping both Franz and Allegro versions of certain code was necessary. Often it is simply easier to write Allegro code from scratch than to rewrite Franz code to be compatible with both Lisps. For example, the paradigms for dealing with strings and file input/output differ greatly between the two Lisps. Conditional compilation was used to provide dupli-

cation in these situations. Throughout the rest of this paper areas of the port which utilized this translation technique are noted.

6 Characters/Fixnums

Franz characters are represented as fixnums. COMMON LISP, on the other hand, has a separate character type. An examination of the Franz system revealed that character/fixnum objects were used in two distinct fashions: as a representation for keystrokes and in comparison operations with the cursor character. This section explains how the implementation was changed to allow COMMON LISP characters to be used in both of these instances.

6.1 *Pan*'s Internal Character Representation

Before going on to discuss keystrokes and the cursor character it is important to understand what keeps these objects separable. At the very lowest level of *Pan*'s text representation are vectors of 16-bit objects. Each 16-bit value represents a single character. Eight bits are used for the ASCII value of the character and four bits each are used for the mode and font information. Keystrokes are the user's input to the system and are converted to the internal representation upon insertion into a buffer. The cursor character can be examined with the function `Cursor-Character`. This function converts the internal 16-bit representation to a standard character object. Conversions in both directions are only concerned with the eight-bit ASCII codes. No comparisons based on the mode or font information are supported, so these bits can be ignored. Furthermore, none of the Lisp code places any interpretation on the mode or font bits. The C code of *Pan* is solely responsible for setting and examing these bits. More will be presented on this topic during the discussion of the foreign function interface to C.

6.2 Keystrokes

Keystrokes are *Pan*'s internal representation for keyboard events. Objects of this type are created when the user presses a key or when the system parses a user's keystroke specification. These specifications are strings used in setting key bindings. Three types of conversions are used in manipulating keystroke objects. In the first case keystrokes are converted to the specification string format for announcement to the user, as in the *Emacs* minibuffer. The second conversion maps keystrokes to the internal character representation.

This occurs as characters are inserted into a buffer using `Self-Insert`. Lastly, keystrokes are converted to fixnums based on their ASCII values. The command dispatch mechanism then uses these fixnums for access into a keymap vector.

The following is a list of the changes needed to switch the keystroke representation from character/fixnum to the COMMON LISP character:

- Franz macros, such as `char=`, `graphic-char-p` and `char-code`, provided an abstraction for treating character/fixnum objects as characters. These macros were moved to the extension package to avoid collision with Allegro counterparts. See Appendix A for the macro definitions which provided the character abstraction.
- Allegro functions for mapping between keystrokes specifications and keystroke objects were written and introduced into the system with conditional compilation.
- The low level macro for mapping a COMMON LISP character to the *Pan* character representation was also introduced with conditional compilation.
- Existing Franz code was rewritten to include the necessary conversions between characters and fixnums for keymap accessing. This involved no more than adding calls to `char-code` and `code-char` in appropriate places. No conditional compilation was needed here, since these macros were available in the Franz extension package. Of course, they did nothing more than extract a fixnum from a fixnum or coerce a fixnum to a fixnum, but this was enough to allow the same source code to function in both Lisps.
- Finally, the C routine which reports keyboard events to Lisp was altered to pass COMMON LISP characters to the Allegro system. Conditional compilation allowed Franz to continue receiving character/fixnum objects. See the section on the foreign function interface for a description of these changes.

6.3 The Cursor Character

Editing operations move the cursor from place to place within a buffer and will occasionally need to know what character is beneath it. As mentioned above, the only access to the character under the cursor is through the function `Cursor-Character`. This function is responsible for converting

the *Pan* character to a representation which the user may examine. Once in that representation there are two common uses for the object. In the first case the syntax class of the character is retrieved for determining if a special action is needed. For example, when a character of class `:rbracket` is inserted into a buffer, the matching left bracket may be shown to the user. The second situation arises when a character is tested for membership in the string `:indentation-chars`.

Switching the Allegro representation of the cursor character to a character object required the following changes:

- The low level macro for mapping a *Pan* character to the COMMON LISP character representation was introduced with conditional compilation.
- In order to retrieve the syntax class of a character its ASCII value must be extracted for indexing into a vector. Functions for performing this operation were altered to use `char-code` on the character arguments. In some areas where efficiency was a problem, the bits for the ASCII value were taken directly from the low level representation to avoid the conversion to a COMMON LISP character.
- The final step was to use the function `position` instead of `char-index` to check character membership in the string `:indentation-chars`. A macro defining `position` in terms of `char-index` was also added to the Franz extension package.

7 Strings

The most common way to deal with strings in Franz is to perform operations on lists of characters, which are of course just lists of fixnums. Objects of type string have appeared in later releases, but the majority of our system was built before these capabilities existed. Even with the availability of a few string operations in Franz, it is clear that the paradigms for string manipulations between the two Lisps are quite different. COMMON offers a wide range of string manipulations, whereas Franz relies heavily on a few primitive functions. For this reason almost all of the string handling code had to be rewritten in Allegro. The following two sections explain how the four operations `implode`, `exploden`, `tconc` and `lconc` were eliminated from the system and replaced with COMMON LISP string operations.

7.1 Implode and Explode

In Franz an object of type string exists primarily in three forms: as a symbol whose print name is the desired string, as a sequence of characters surrounded by double quotes and as a list of fixnums representing the ASCII value of each character. While the first two representations offer a means for input and output of strings, manipulating these objects in more interesting ways requires that the they be "exploded" into the list form. Once in that form list operations can be applied. The functions **exploden** and **implode** are used to convert the symbol to the list and the list to the symbol, respectively[5]. Our Franz implementation relied heavily on this paradigm for searching strings and translating them back and forth between internal representations and user readable form. To illustrate the use of these functions in our system some examples follow. The code below was not taken directly from *Pan*, but has been written to demonstrate the kinds of manipulations we perform on strings.

Example 1: Searching strings for breakpoints is an important functionality used by the apropos facility and the file system interface. The following Franz function will search a string or symbol for the final occurrence of the character "/" and return a symbol composed of all subsequent characters:

```
(defun from-final-slash (string-or-symbol-arg)
  ;; Bind reverse-list to the destructively modified
  ;;  reverse of the exploded list of ASCII values.
  (let ((reverse-list
         (nreverse (exploden string-or-symbol-arg))))
    ;; Use memq to find the first location of the slash.
    ;;  This is the last location in the unreversed list.
    ;;  Difference reverse-list with the portion containing
    ;;  the slash to extract the desired sublist.  Lastly,
    ;;  destructively reverse the sublist and implode the
    ;;  result, returning the symbol.
    (implode (nreverse (ldiff
                        reverse-list
                        (memq #\/ reverse-list))))))
```

In Allegro the following will perform the same function with both the argument and return value as strings:

[5]There are many more "plode" functions. The variations allow control over how special characters are handled and whether the return values contain symbols or strings. The two given here will suffice for this discussion.

```
(defun from-final-slash (string-arg)
  ;; Bind slash-pos to the position of the final
  ;;  slash within string-arg or nil if not found.
  (let ((slash-pos
         (position #\/ string-arg :from-end t)))
    ;; If a slash was found return the string containing
    ;;  all characters after its position; otherwise
    ;;  return the entire string.
    (if slash-pos
        (subseq string-arg (1+ slash-pos))
        string-arg)))
```

Example 2: In many places throughout the system a string must be translated from a user readable form to an internal form or vice versa. Functions of this sort iterate over the characters of the string building the new representation. The Franz function below transforms a string or symbol into a list. Ranges, indicated by a dash between two characters, are consed into a dotted pair of ASCII values. All other characters are simply transformed to ASCII and appended to the list. For example, `"a-fqr-vc"` becomes `((97 . 102) 113 (114 . 118) 99)`. Mappings of this kind are used to parse the user's specification for setting character classes and key bindings. Reverse mappings also exist for describing these settings to the user.

```
(defun parse-key-seq-specification (string-or-symbol-arg)
  (do ((exploded-list (exploden string-or-symbol-arg))
       ;; Initialize the tconc structure.
       (result-char-tconc (ncons nil)))
    ;; Iterate until the list is empty.
    ((null exploded-list) (car result-char-tconc))
    (cond
     ;; If the remaining list is >= 3 elements long and the
     ;;  second element is a dash then a range is specified.
     ((and (>= (length exploded-list) 3)
           (= (cadr exploded-list) #\-))
      ;; Append the dotted pair of ASCII values to the
      ;;  tconc structure and rebind exploded-list to
      ;;  skip over the processed elements.
      (tconc result-char-tconc
             (cons (car exploded-list) (caddr exploded-list)))
      (setf exploded-list (nthcdr 3 exploded-list)))
     ;; otherwise just append the value.
     (t (tconc result-char-tconc (car exploded-list))
        (setf exploded-list (cdr exploded-list))))))
```

The Allegro version of this function produces a list of characters (not fixnums) that are in reverse order. A final reversing of the elements of the list is unnecessary, since only the range information need be preserved. For example, "a-fqr-vc" becomes (#\c (#\r . #\v) #\q (#\a . #\f)).

```
(defun parse-key-seq-specification (string-arg)
  (let ((len (length string-arg)))
    (do ((i 0 (1+ i))
         (result-char-list nil))
      ;; Iterate until the end of the string.
      ((= i len) result-char-list)
      (let ((curr-char (schar string-arg i)))
        (cond
         ;; If the end of the string is at least
         ;;  2 positions away and the next character
         ;;  is a dash then a range is specified.
         ((and (< (+ 2 i) len)
               (char= (schar string-arg (1+ i)) #\-))
          ;; Prepend the dotted pair of characters to
          ;;  the list, incrementing i to skip over
          ;;  the processed characters.
          (push (cons curr-char
                      (schar string-arg (incf i 2)))
                result-char-list))
         ;; otherwise just prepend the character.
         (t (push curr-char result-char-list)))))))
```

In order to remove exploden and implode, conditional compilation was used to install Allegro versions of code that handle string processing. Although writing implode and explode in Allegro would have been trivial the functionality involving "plodes" was eliminated from the system. This meant more work, but writing Allegro code to behave like Franz was not the goal of the port.

7.2 Tconc and Lconc

Closely related to the problem of "plodes" in *Pan* were the functions tconc and lconc. They operate on something known as a tconc structure, a cons whose car is a list and whose cdr is the last element of that list. These operations can be used to construct a list quickly by appending elements directly onto it as shown by the second example above. A single element is appended with tconc, while lconc appends an entire list splicing in the

elements. When all of the elements have been tconced onto the structure, taking its car yields the desired list.

In situations like the one in example 2, Allegro code was written to use push for building a list based on some data. If preservation of the order was necessary a final **reverse** was applied. Other uses of these functions, however, were not intended to build a list at all, but were designed to eventually yield a string. In these cases Franz functions would build up a list with tconc and then apply "imploding" functions to force the list into its string form. Allegro counterparts to these functions were written using string concatenation as shown in the following example.

Example 3: The following Franz function translates a Unix file expansion regular expression (containing "*" and "." only) into a form acceptable to ed. For example "*.cl" becomes "[^#\space #\tab][^#\space #\tab]*\\.cl". This code is used to provide matching of file names for automatic execution and loading mechanisms. The translation is necessary since *Pan* uses the Unix regular expression search facility which requires an ed format.

```
(defun filename-to-re (string-arg)
  (let ((result-tconc (ncons nil)))
    ;; Iterate until the end of the string returning
    ;;  the resulting list imploded as a string.
    (dotimes (i (string-length string-arg)
                (implodes (car result-tconc)))
      ;; Bind curr-char to ASCII value of the i-th character.
      (let ((curr-char (getcharn string-arg (1+ i))))
        ;; Switch on curr-char, tconc-ing onto result-tconc.
        (case curr-char
          (#\* (tconc result-tconc #\[)
               (tconc result-tconc #\^)
               (tconc result-tconc #\space)
               (tconc result-tconc #\tab)
               (tconc result-tconc #\])
               (tconc result-tconc #\[)
               (tconc result-tconc #\^)
               (tconc result-tconc #\space)
               (tconc result-tconc #\tab)
               (tconc result-tconc #\])
               (tconc result-tconc curr-char))
          (#\. (tconc result-tconc #\\)
               (tconc result-tconc curr-char))
          (t   (tconc result-tconc curr-char)))))))
```

Concatenation of strings provides equivalent functionality in Allegro:

```
(defun filename-to-re (string-arg)
  (let ((result-string ""))
    ;; Iterate until the end of the string
    ;;   returning the translation.
    (dotimes (i (length string-arg) result-string)
      ;; Bind curr-char to the i-th character.
      (let ((curr-char (schar string-arg i)))
        ;; Build up the new string by concatenating the result
        ;;  so far with a string based on curr-char.
        (setf result-string
              (concatenate
               'string
               result-string
               (case curr-char
                 (#\* (format nil "[^ ^A][^ ^A]^C"
                              #\tab #\tab curr-char))
                 (#\. (format nil "\\^C" curr-char))
                 (t   (format nil "^C" curr-char)))))))))
```

Notice that with the addition of Franz macros for taking string lengths, accessing characters and concatenating strings, the above Allegro function could be compatible with both Lisps. However, since the Franz function is more efficient, conditional compilation may be more desirable. This decision will depend on the application. Currently, *Pan* has both Allegro and Franz versions of the code that performs regular expression translation.

8 Foreign Function Interface

In order to achieve the efficiency necessary within an editing environment *Pan* has an extensive foreign function interface to C. The C routines handle such things as the window system and file writing operations. Our system uses Lisp-to-C and C-to-Lisp calls. Setting up the framework for the interface was not difficult after the Allegro syntax was known; however, getting C and Lisp to interpret arguments and return values correctly was one of the most troublesome areas of the port. Because this is another area for which the paradigms in Franz and Allegro differ greatly, conditional compilation was used in both the Lisp and C worlds.

8.1 Lisp to C

The Lisp code of *Pan* uses well over 100 functions provided by C. These functions:

- provide a layer of abstraction between *Pan* and the window system,
- alter character font and mode information,
- read, write, append and copy files,
- provide functionality needed in lexical analysis, and
- perform regular expression searching.

The majority of these functions (over 80) are dedicated to providing the window system abstraction.

Defining the necessary interfaces in Allegro required moving the function binding mechanism from C to Lisp and adding declarations of the argument and return types. Fortunately, these simple syntactic changes resulted in the correct installation of about 90% of the Lisp-to-C calls. This included almost all of the window code and about half of the routines which provided the other services. Calls falling into this category pass fixnums, strings or simple arrays and return an integer or no value at all. Because the foreign function interface handles the translation between the Lisp and C representations of these types of objects, the code on either side of the interface required no semantic changes. The remaining 10% of Lisp-to-C calls were more complicated, performing tasks requiring special types of arguments and return values. These are explained below.

8.1.1 Traversing the Text Representation

Character painting, file writing, setting font and mode information, performing lexical analysis and regular expression searching all require traversing *Pan*'s internal representation of text. Fortunately, Franz-C code had been written for these purposes when *Pan* was originally developed. The basic algorithms for iterating over the Lisp text representation had already been debugged. Only the methods for converting objects to a C-readable form had to be installed.

Data representation in Franz and C is quite similar, so few conversions are necessary for the proper interpretation of foreign objects. However, Allegro and C representations differ significantly. For this reason macros for extracting C-readable data from Lisp structures are provided in the library

file *lisp.h*. By defining our own accessor macros in terms of those in *lisp.h* we were able to create Allegro-C code for traversing the text representation. Unfortunately, debugging the C routines was hampered by a lack of documentation for the kinds of manipulations we were performing. When *lisp.h* failed to provide an answer we would resort to examing a structure with the Allegro inspector or even printing a raw hexadecimal dump of the structure from C. After some trial and error we were able to converge to the correct C structure definitions and macros.

8.1.2 Fill Pointer Arrays

During lexical analysis and regular expression searching, fill pointer arrays are used by Lisp as variable length storage for strings. C is responsible for copying character values into the array as it traverses the Lisp text representation. Several interesting problems were encountered while debugging these routines.

In regular expression searching the string is used to hold a line of text copied from within a buffer. The fill pointer array is created in Lisp and passed to C for destructive modification. The foreign function interface does not just pass the raw Lisp value, but rather, a valid character pointer, which can then be used to fill in the characters of the string. When control returns to Lisp the fill pointer is set to the location of the null character provided by C. This interface was relatively easy to get correct, especially when compared to the lexer.

The interface to the lexer in *Pan* uses a slight variation of the above scheme. Instead of passing the array across the interface once per lexeme, the array is created in Lisp and accessed through the function `lisp_value()`. This function is provided as part of the Allegro/C foreign function interface for direct manipulation of Lisp objects. It offers none of the argument checking or translation that occurs when values are passed into C. Therefore, routines manipulating the array in this case had to extract the character pointer using pointer arithmetic on the Lisp value. No macros for performing these calculations are available in *lisp.h*, so they had to be written from scratch by inspecting the Lisp structures in Allegro.

8.2 C to Lisp

Although *Pan* is written primarily in Lisp, events from the window system are captured in the C code. Lisp handlers are then called to process them.

Porting these interfaces required using conditional compilation to introduce `defun-c-callable` forms of the function definitions for Allegro. C-callable functions are identical to normal functions, except that the types of the arguments must be declared. Once the Allegro functions were C-callable the actual calls were added to C using the library function `lisp_call()`.

The final change necessary was to alter the C event dispatcher to call the Lisp keyboard handler with a COMMON LISP character object. Again, *lisp.h* provided no help, so the macro was written to perform the necessary conversion. Originally it was written to produce the character representation explained in *lisp.h*, however this documentation proved to be wrong. Only by using the Allegro inspector was the correct character representation discovered.

8.3 Other Interface Problems

Many of the arguments passed across the foreign function interface in *Pan* are boolean values. In Lisp true and false are represented by non-`nil` and `nil` respectively, while in C non-0 and 0 are used. This presents a problem for functions using booleans, since the Allegro interface provides no translation for these values. Two macros were written to solve this problem until the interface is improved. The first translates Lisp boolean arguments to C and the second translates C boolean return values to Lisp. For example, the form `(boolean-hack-result foo fix-foo)` is a macro call that defines `foo` to be a function which invokes `fix-foo` (a C routine) and then translates the return value to a Lisp boolean. A similar macro call sets up the translation for arguments passed to C. These macros can be easily removed when Allegro augments the foreign function interface to include boolean translation. No other corrections to the system will be necessary.

The final problem encountered in porting the C/Lisp interface was the Allegro garbage collector. Franz had a simple garbage collection (gc) scheme in which values are never moved once created. Allegro, however, does not make the same guarantee. If C has a pointer into the Lisp world and garbage collection occurs the pointer may be invalidated. To get around this, Allegro has a system in which values may be registered in Lisp and accessed in C with calls to `lisp_value()`. Fortunately, *Pan* had been written so that C keeps very little information about the Lisp world. In fact the only area of the system that required registration of values was the lexer. The lexer caches pointers to the first and last text nodes of the buffer region currently being analyzed, as well as to the fill pointer array mentioned above. To

avoid any problems the C gc-after hook was used to update the caches after every gc with new calls to `lisp_value()`.

9 System Enhancements

9.1 &prompt Argument Types

The *Pan* command definition facility provides the lambda keyword `&prompt` in addition to those normally provided by Lisp. It allows command definitions to include arguments which receive default bindings by prompting the user. For example:

```
(Define-Command Echo
  (&prompt (user-input "Enter string:"))
  :help "A simple command to echo the user input
         to the Annunciator line of the active viewer."
  (Announce "You typed: ~S" user-input))
```

defines a simple command that echoes a string input by the user. During the port the syntax of `&prompt` was expanded to include a type specification for each argument. The above definition would now be written as follows:

```
(Define-Command String-Echo
  (&prompt (user-string string "Enter string:"))
  :help "A simple command to echo the user input string
         to the Annunciator line of the active viewer."
  (Announce "You typed: ~S" user-string))
```

This allows arguments to have different (user defined) prompters automatically associated with them through the command definition. The argument list

```
(&prompt (arg1 type1 string1) (arg2 type2 string2) ...)
```

is expanded by the command definition mechanism to

```
(&optional (arg1 (prompter-type1 string1))
           (arg2 (prompter-type2 string2))
           ...)
```

Because `&prompt` expands to `&optional` only one of the two may appear in an argument list. Also, since `&optional` is required to be the first lambda keyword in the list, `&prompt` carries the same restriction.

Typing of the `&prompt` arguments is handy for allowing the user to quickly integrate a new prompter into the system. Adding a prompter simply

requires defining it and augmenting the list of prompters and types. In fact, this is precisely how pathname objects were added to *Pan*. When **&prompt** functionality was improved, pathname command arguments were identified and tagged with the type **pathname**. This new type was then temporarily associated with the prompter for strings. Later when the **Prompt-For-File** command was added, the function which associates a prompter with a type was changed. With this single alteration all commands which called for the user to enter a pathname automatically became clients of the new prompter.

9.2 File System Interface

Pan as originally implemented in Franz was tied very closely to the underlying file system. This is not surprising since file handling in Franz is modeled after the paradigm used by C. Simply porting this code in a purely syntactic manner to Allegro would have been foolish given the existence of COMMON LISP pathnames. These objects help to eliminate an application's dependency on a file system, by denoting files in an operating system independent way. It is, of course, impossible to completely ignore the file system, but abstracting away as much as possible is desirable. This is especially true for *Pan* which is concerned with the editing and presentation of structured objects. As other repositories besides files become available *Pan* will want to take advantage of them as well.

Removing our strong dependency on the file system required carefully defining the role and use of directories, files, file names and buffer names[6]. Unfortunately the boundaries between these objects were extremely blurred by their being used almost interchangeably throughout the code. When the user specified a file with a string it became the file name and the buffer name. Any of the three values could be used in place of any other. The rewrite of the file system interface enforces the following distinctions between these objects:

- *file* - a pathname used to denote the physical file. This is strictly an internal representation for file specifications.
- *directory* - a pathname used to denote a directory within the file system. A directory is part of the working environment for commands executed from within a buffer. Every buffer has its own working directory.

[6]Changes to *Pan*'s file handling were based on design notes by Michael Van De Vanter.

- *file name* - a string used to communicate with the user about files. A file name is a function of the file (specification) and exists only to provide the user with a readable form of the internal pathname representation.
- *buffer name* - a unique string used to identify a memory resident object which may be viewed and possibly edited. Notice that this implies a buffer need not have any secondary storage associated with it.

Distinguishing the role of these objects required rewriting code in four basic areas of *Pan*: buffer management, buffer commands, file input/output and file commands. Pathnames were implemented as strings in the Franz version and a few of the basic manipulations were installed using macros. Section 2 of Appendix A contains the macro definitions used to extend Franz. As usual, conditional compilation allowed divergent code to exist in both Lisps.

9.2.1 Buffer Management

The buffer structure was expanded to include three new slots: name, file and directory. The existing file name slot remained only as a printable representation of the buffer file specification. Code which had previously manipulated the buffer file name was changed to obey the conventions given above. Paramount among these changes was the elimination of the ability to identify a buffer by its file name. Buffers are now identified by their name and if necessary can also be located through their file. The latter case arises only when checking to see if a file is being edited before retrieving it from the file system. Three distinguished buffers, the help buffer, base buffer and currently active buffer, can also be accessed through special macros.

9.2.2 Buffer Commands

Buffer commands provide such operations as setting the active buffer, allocating viewports to buffers and saving the files associated with them. Commands were rewritten to enforce proper usage of buffer slot information. Also, commands for creating unique buffer names and scratch buffers were added. The latter is a demonstration of *Pan*'s new independence from the file system by providing an editable object that is not associated with any secondary storage. Finally, now that each buffer had its own working directory, a command for setting this slot was added and the machinery for providing a single global working directory was removed.

9.2.3 File Input/Output

The input/output module of *Pan* makes extensive use of file specifications to perform tilde expansion, check file properties, retrieve user information and open and close files for reading and writing. It generally serves as a layer of abstraction on top of the operating system. Conditional compilation was used extensively in this area to include Allegro versions of file manipulation functions. Also, during work on this part of the system it was discovered that the algorithm used by *Pan* for writing buffer files was unnecessarily slow. To write a file the following steps were taken:

1. Open the file in Lisp.
2. Pass C a file pointer and a node of the text representation for writing. pointer.
3. Write the characters contained in the node to the file.
4. If the end of the buffer has been reached go to 6.
5. Go to 2.
6. Close the file in Lisp.

This code had been written very early in *Pan*'s development and did not take advantage of the ability of C routines to traverse the text representation. The code was rewritten to use the following algorithm (in Allegro only):

1. Pass C a region of the buffer and the name of a file to write to. A region is specified as a beginning and ending node in the text representation.
2. Open the file.
3. C traverses the text representation writing all characters within the region.
4. Close the file and return control to Lisp.

9.2.4 File Commands

File commands provide a second layer of abstraction on top of file input and output that allow the user to perform high level file manipulations. These include operations such as visiting, loading, writing, backing-up and check-pointing files and creating file names from files. Besides altering commands to enforce proper usage of the buffer slot information, several enhancements were added as well.

The command `Prompt-For-File` was written and associated with the `&prompt` type `pathname`. This new prompter primes the user's input string with the value of the active buffer directory just as in the *Emacs* minibuffer. The string is then tilde-expanded and merged with the buffer directory, ensuring that a complete absolute pathname is created.

`List-Files`, the command for listing a directory in the help buffer, was changed to set the buffer working directory. Files selected from the listing can now be visited with no further specification.

`Visit-File` was changed to accept an optional argument specifying a read-only visit. The semantics of a read-only were then refined to ensure that no backing-up or checkpointing of the file would occur.

9.3 Help System

Pan has an elaborate help facility which automatically documents commands added to the system. The user gains access to the documentation with apropos and describe facilities. Until recently the apropos mechanism distinguished the singular and plural forms of its arguments. For example, requesting apropos information for the keyword "files" would ignore any entries under "file". The mechanism was altered to look for both entries regardless of which keyword was supplied.

Another weakness appeared in the description mechanism. When a user asked for information about a command, the system would print its documentation string, the arguments it took and whether or not it could be bound to a menu or a key. However, no information was given about what those current bindings may have been. The only way to discover the key or menu binding of a command was to list all of the bindings. Since there are not that many menu bindings, the user could look through the list quickly. If that was unacceptable the menus themselves could always be explored. Looking through the list of key bindings, however, was quite a chore. There are over 100 default bindings alone. To eliminate the need for examining the entire binding list to extract a single entry two commands were added. The first maps key sequence specifications to commands and the second performs the inverse operation. This information is made available through the describe facility or may be accessed by invoking the mapping commands directly.

10 Functionality Loss

This section describes functionality of the Franz system that could not be fully supported with Allegro.

10.1 Error Handling

Error handling within *Pan* occurs at both the user level and internally. User errors are detected by the system and signaled with the command `Editor-Error`. These are quite common and do not interrupt execution. `Editor-Error` simply terminates execution of the current command, prints a message and returns control to *Pan*'s top level. Errors occurring at the internal level are caused by mistakes in the system code. These would normally fall through to the underlying Lisp and put the user into a break loop. To capture errors of this type the Franz version of *Pan* rebinds the Lisp error handlers `%ERall` and `%ERtpl`. At the time an internal error occurs, our functions are called with arguments explaining the problem. These functions in turn signal the error to the user and perform any recovery necessary. If recovery is impossible then the user is advised to reset the system.

Pan makes use of this error handling technique in three situations:

1. `%ERall` is bound to load-error-handler while a file is being loaded into the running system using the command `Load-File`.
2. `%ERall` is bound to exec-error-handler during execution of the command `Execute-Lisp-Line`, which takes a Lisp form entered by the user and evaluates it.
3. `%ERtpl` is bound to pan-error-handler at start-up to trap internal errors. This allows the user to be signaled from within the system itself instead of entering a Lisp break loop.

Allegro does not provide a facility for rebinding error handlers. The first two cases above have been approximated by using `excl:errorset` to wrap the `load` and `eval` forms explicitly. This will effectively trap any errors and allow the user to be signaled. It is only an approximate solution since the cause of the error is only known by the Lisp system and not by *Pan*. For example, during loading it is not possible to detect an error caused by a nonexistent file from one caused by a problem during the load itself.

The third case above cannot be so easily approximated. It differs from the first two situations in that no particular form can be wrapped with

`excl:errorset` to trap an internal error. An error of this sort can happen anywhere throughout the entire system. For now this problem has been deferred until Allegro is expanded to include a mechanism similar to that of Franz. Until that time users will find themselves entering break loops when internal errors occur.

10.2 GC Hooks

Franz provides both before and after hooks into its garbage collection mechanism. In *Pan* these hooks are used to provide a facility for registering functions to be executed before and after gc. The most common usage of this facility is to change the mouse cursor to signal the user when a gc is happening. In Allegro only a gc-after hook exists in Lisp, while both gc-before and gc-after hooks exist for C foreign functions. Although it would be possible to build the registration mechanism utilizing the C hooks, it would be best to wait for Allegro to include them explicitly in Lisp.

11 Porting Outcomes

11.1 System Size

When the port began there were about 18,000 lines of Franz Lisp source code. It is interesting to note that this number has remained almost unchanged as a result of the port. Two different implementations of the system now exist in the space it previously took to store one.

A more significant change can be seen by looking at the size of the system binaries. The start-up runtime image of the Franz system is 4.3 megabytes, while the Allegro image is 5.5 megabytes. This can be attributed to both the added functionality provided in COMMON LISP and the fact that the Allegro compiler is part of the environment and not a stand-alone program as in Franz.

11.2 Performance

Currently the system is being run in Allegro with all debugging features turned on to catch any possible problems. This includes argument checking at the foreign function interface, printing diagnostic messages and checking invariants. When these are removed the system is expected to exhibit better performance than the Franz version with the following exceptions:

- File loading in Allegro is much slower due to the extra checking performed by the loader.
- Allegro does not provide a quick way to remove C entry points from a running Lisp. *Pan* needs this capability when new language tables are being loaded.
- As mentioned earlier, the fixnum operations within the system use general mathematical functions. These run slower than the fixnum specific operations provided in Franz.
- The garbage collector will have to be tuned for our system.

11.3 Writing Portable Lisp Systems

There are two pieces of advice that I would offer to programmers writing portable Lisp systems. The first would be to avoid keeping pointers into Lisp from foreign code. Pointers of this sort greatly reduce portability by requiring implementation dependent manipulation to avoid garbage collection problems. The second would be to use abstract data types to separate different uses of the same base type. For example, in *Pan* we used a character abstraction to distinguish fixnums as numbers from fixnums as characters. ADT's will help to eliminate many of the problems encountered when moving from a weak type system to a stronger one.

12 Conclusion

This document has explained the process of porting *Pan I* from Franz Lisp to Allegro COMMON LISP. By using a combination of code rewriting, macro introduction and conditional compilation the Franz sources of the system have been transformed to be compatible with both Lisps. The success of this technique has shown that an existing system can be ported without leaving it nonfunctional for long periods of time.

13 Acknowledgements

I would like to thank my advisor, Susan L. Graham, and all the members of our research group for their help and support. Special thanks go to Chris Black for helping with testing and to Jacob Butcher and Michael Van De

Vanter for their help and patience during enumerable debugging sessions and design meetings.

References

[1] *Allegro* COMMON LISP *User Guide*, Franz Inc., January 1988.

[2] Robert A. Ballance and Michael L. Van De Vanter. *Pan I: An Introduction for Users.* Technical Report 88/410, Computer Science Division, UC Berkeley, March 1988.

[3] Robert A. Ballance, Michael L. Van De Vanter and Susan L. Graham. *The Architecture of Pan I.* Technical Report 88/409, Computer Science Division, UC Berkeley, August 1987.

[4] *Franz Lisp Reference Manual*, Franz Inc., March 1987.

[5] Guy L. Steele Jr., COMMON LISP*: The Language*, Digital Equipment Corporation, 1984.

A Macro Extensions

A.1 Aliases defined with defnewname

Allegro	*Franz*
`array-total-size`	`vsize`
`copy-tree`	`copy`
`copy-list`	`copy`
`aref`	`vref`
`floor`	`*quo`
`force-output`	`drain`
`symbol-function`	`getd`
`excl::pointer-to-fixnum`	`maknum`
`excl:fixnump`	`fixp`
`excl:if*`	`if`

A.2 General Macros

```
(defmacro defconstant
          (symbol value &optional (documentation nil doc?))
  (if doc?
      `(eval-when (compile load eval)
         (putprop ',symbol ,documentation 'documentation)
         (defvar ,symbol ,value))
      `(eval-when (compile load eval)
         (defvar ,symbol ,value))))

(defmacro defparameter
          (symbol value &optional (documentation nil doc?))
  (if doc?
      `(eval-when (compile load eval)
         (putprop ',symbol ,documentation 'documentation)
         (defvar ,symbol ,value))
      `(eval-when (compile load eval)
         (defvar ,symbol ,value))))

(defmacro defnewname (new-function-name old-function-name)
  `(defmacro ,new-function-name (&rest args)
     `(,',old-function-name ,@args)))

(defmacro byte (size position)
  `(list :byte ,size ,position))

(defmacro byte-position (byte)
  `(third ,byte))

(defmacro byte-size (byte)
  `(second ,byte))


(defmacro ldb (bytespec integer)
  (let ((byte-spec (car (errset (eval bytespec)))))
    (if (and bytespec (eq (first byte-spec) :byte))
        (let ((position (byte-position byte-spec))
              (size (byte-size byte-spec)))
          (let ((byte-mask (lsh (lognot (lsh -1 size)) position)))
            `(lsh (logand ,integer ,byte-mask) ,(- position))))
        (cli:error "%ldb called with illegal bytespec argument."))))
```

```
(defmacro dpb (newbyte bytespec integer)
  (let ((byte-spec (car (errset (eval bytespec)))))
    (if (and bytespec (eq (first byte-spec) :byte))
        (let ((position (byte-position byte-spec))
              (size (byte-size byte-spec)))
          (let ((byte-mask (lsh (lognot (lsh -1 size)) position)))
            `(logior (logandc2 ,integer ,byte-mask)
                     (logand (lsh ,newbyte ,position) ,byte-mask))))
        (cli:error
          "%dpb called with illegal bytespec arguments."))))

;;; This macro does not support fill pointers.
(defmacro make-array (dimension &key (element-type t)
                                     (initial-element nil)
                                     (fill-pointer nil))
  (multiple-value-bind (success? type)
                       (errorset (eval element-type))
    (if success?
        (cond ((equal type 't)
               `(new-vector ,dimension ,initial-element))
              ((equal type 'fixnum)
               `(new-vectori-long ,dimension))
              ((equal type 'string-char)
               `(new-vectori-byte ,dimension))
              ((equal type '(unsigned-byte 32))
               `(new-vectori-long ,dimension))
              ((equal type '(unsigned-byte 16))
               `(new-vectori-word ,dimension))
              ((equal type '(unsigned-byte 8))
               `(new-vectori-byte ,dimension))
              ((equal type '(signed-byte 32))
               `(new-vectori-long ,dimension))
              ((equal type '(signed-byte 16))
               `(new-vectori-word ,dimension))
              ((equal type '(signed-byte 8))
               `(new-vectori-byte ,dimension))
              (t (cli:error
                   "%Error: make-array element type unknown")))
        (cli:error
          "%Error: make-array type argument evaluation error"))))

(defun integerp (sexpr)
  (or (fixp sexpr)
      (bigp sexpr)))
```

```
(defun characterp (sexpr)
  (fixp sexpr))

(defmacro merge (result-type seq1 seq2 comp-pred)
  `(insert (car ,seq1) ,seq2 ,comp-pred t))

(defmacro namestring (pathname)
  `(string ,pathname))

(defun string-upcase (string)
  (string (maknam (mapcar #'char-upcase
                          (aexploden string)))))

(defun string-downcase (string)
  (string (maknam (mapcar #'char-downcase
                          (aexploden string)))))

(defin schar (string index)
  (getcharn string (1+ index)))

(defmacro <=! (a b &optional c)
  (if c
      `(let ((%temp ,b))
         (and (<=& ,a %temp)
              (<=& %temp ,c)))
      `(<=& ,a ,b)))

(defmacro <! (a b &optional c)
  (if c
      `(let ((%temp ,b))
         (and (<& ,a %temp)
              (<& %temp ,c)))
      `(<& ,a ,b)))


(defmacro char-code (char)
  `(logand ,char #xff))

(defmacro code-char (char)
  `(logand ,char #xff))
```

```
(defun standard-char-p (char)
  (let ((code (char-code char)))
    (and (eql code char)
         (or (<! 31 code 127)
             (= code #\newline)))))

(defun graphic-char-p (char)
  (<! 31
      (char-code char)
      127))

(defun string-char-p (char)
  (<! -1
      (char-code char)
      256))

(defun alpha-char-p (char)
  (let ((m (char-code char)))
    (or (<=! #\A m #\Z)
        (<=! #\a m #\z))))

(defun upper-case-p (char)
  (<=! #\A
       (char-code char)
       #\Z))

(defun lower-case-p (char)
  (<=! #\a
       (char-code char)
       #\z))

(defun digit-char-p (char &optional (radix 10))
  (unless (<=! 0 radix 36)
    (error "~S is an illegal input radix."  radix))
  (let* ((code (char-code char))
         (value (cond ((<=! #\0 code #\9) (- code #\0))
                      ((<=! #\A code #\Z) (- code (- #\A 10)))
                      ((<=! #\a code #\z) (- code (- #\a 10))))))
    (if (<! value radix)
        value
        nil)))
```

```
(defun both-case-p (char)
    (let ((m (char-code char)))
    (or (<=! #\A m #\Z)
        (<=! #\a m #\z))))

(defun directory-namestring (name)
  (let ((rname (nreverse (aexploden (tilde-expand name)))))
    (implodes (nreverse (memq #\/ rname)))))

(defun alphanumericp (char)
  (let ((m (char-code char)))
    (or (<=! #\0 m #\9)
        (<=! #\A m #\Z)
        (<=! #\a m #\z))))

(defmacro char= (c1 c2)
  `(=& ,c1 ,c2))

(defmacro char/= (c1 c2)
  `(not (char= ,c1 ,c2)))

(defmacro char< (c1 c2)
   `(<! ,c1 ,c2))

(defmacro char<= (c1 c2)
   `(<=! ,c1 ,c2))

(defmacro char> (c1 c2)
   `(<! ,c2 ,c1))

(defmacro char>= (c1 c2)
   `(<=! ,c2 ,c1))

(defun char-upcase (char)
  (if (lower-case-p char)
      (logxor char 32)
      char))

(defun char-downcase (char)
  (if (upper-case-p char)
      (logxor char 32)
      char))
```

```
(defmacro throw (tag result)
  `(let ((%tag ,tag)
         (%result ,result))
     (when *throw-hook*
       (funcall *throw-hook* %tag %result))
     (lisp:*throw %tag %result)))

(defmacro catch (tag &body forms)
  `(lisp:*catch ,tag
                (progn ,@forms)))

(defmacro tagbody (&body body)
  `(prog ()
         ,@body
         (return nil)))

(defmacro block (name &body body)
  `(prog (,name)
         ,@body
         ,name
         (return ,name)))

(defmacro return-from (block-name &optional (return-form nil))
  `(progn (setf ,block-name ,return-form)
          (go ,block-name)))

(defmacro with-open-file ((stream-var &rest open-args)
                          &body body)
  `(let ((,stream-var (open ,@open-args)))
     (unwind-protect (progn ,@body)
       (when ,stream-var
         (close ,stream-var)))))

(defmacro errorset (form)
  `(let ((%result (errset (multiple-value-list ,form) nil)))
     (if %result
         (apply #'values t (car %result))
         nil)))
```

```
(defun open (pathname &rest options)
  (let ((direction :output)
        (existence :unspecified))
    (do ((options options (cddr options)))
      ((null options))
      (let ((option-arg (cadr options)))
        (case (car options)
          (:direction (setf direction option-arg))
          (:if-exists (setf existence option-arg))
          (:if-not-exists (setf existence option-arg)))))
    (case direction
      (:probe (probef pathname))
      (:input (case existence
                (:unspecified (infile pathname))
                (t (cli:error "%with-open-file :exists error~%"))))
      (:output (case existence
                 (:append (outfile pathname "a"))
                 (:unspecified (outfile pathname))
                 (t (cli:error "%with-open-file :exists error~%"))))
      (t (cli:error "%with-open-file :direction error~%")))))

(defmacro read-char (&optional (input-stream 'piport)
                               (eof-error-p nil)
                               (eof-value nil))
  `(let ((%char (tyi ,input-stream)))
     (if (minusp %char)
         ,eof-value
         %char)))

(defmacro pathname (arg)
  `(string ,arg))

(defmacro peek-char (&optional (peek-type nil)
                               (input-stream 'piport)
                               (eof-error-p nil)
                               (eof-value nil))
  `(let ((%char (tyipeek ,input-stream)))
     (if (minusp %char)
         ,eof-value
         %char)))
```

```
(defmacro read-line (&optional (input-stream 'piport)
                               (eof-error-p nil)
                               (eof-value nil))
  `(let ((%char-list '())
         (%char))
     (while (setf %char
                  (read-char ,input-stream ,eof-error-p))
       (when (eq %char #\newline)
         (return))
       (push %char %char-list))
     (if %char-list
         (values (symbol-name (implode (nreverse %char-list)))
                 (not %char))
         ,eof-value)))

(defun file-namestring (name)
  (let ((tname (nreverse (aexploden (tilde-expand name)))))
    (implodes (nreverse (ldiff tname (memq #\/ tname))))))

(defmacro incf (loc &optional (amount 1))
  `(setf ,loc (+ ,loc ,amount)))

(defmacro decf (loc &optional (amount 1))
  `(setf ,loc (- ,loc ,amount)))

;;; This is a COMMON LISP version of position, except that
;;;  the number returned will be one higher due the start
;;;  position for Franz arrays.o
(defmacro position (element sequence)
  `(char-index ,sequence ,element))

(defmacro read-from-string (str)
  `(readlist (explodec ,str)))

;;; This is a franz definition of CL's assert.  It does
;;;  not support the optional second argument ({place}*).
(defmacro assert (test-form &optional place &rest args)
  ;; don't support CL {place} option, throw it away second arg
  (if args
      `(unless ,test-form
         (cli:error ,@args))
      `(unless ,test-form
         (cli:error "Assertion failure"))))
```