The

ProIcon

Programming Language

for Apple Macintosh Computers

Version 2.0

The ProIcon Group: Salida Colorado and Tucson, Arizona

The ProIcon Programming Language for Apple Macintosh Computers,
© 1989, 1990, 1994 The Bright Forest Company. ISBN 0-939693-08-3.

Version 2.0

Some material in this manual is adapted from the MaxSPITBOL manual with the permission of Catspaw, Inc.

The Bright Forest Company
P.O. Box 12076
Tucson, Arizona
85732-2076

CONTENTS

User's Manual

The Icon Programming Language

Appendices and Index

1

Getting Started

Getting Started

Read this First

Welcome to ProIcon, a powerful programming language that's packed with features to help you solve complex problems quickly.

ProIcon is a high-level, general-purpose programming language that's rich in string and list processing capabilities. ProIcon's novel expression-evaluation mechanism produces quick, neat solutions to even your most complex problems.

ProIcon is an enchanced version of the Icon programming language that was developed at the University of Arizona. ProIcon's added features were designed and implemented by Mark Emmer and Ralph Griswold.

Before You Do Anything Else . . .

Backing Up Your ProIcon

First make backup copies of your original ProIcon disks.

Then install ProIcon on your Macintosh. You'll find complete installation instructions starting on page 1-2.

After that, read **A Guide to the Manual** starting on page 1-3.

Knowing Your Macintosh

You should know generally how to operate your Apple® Macintosh® computer, how to use the mouse and the menus, and how to open documents, copy files, and install applications on your hard disk. If you need help, consult the manuals that came with your Macintosh.

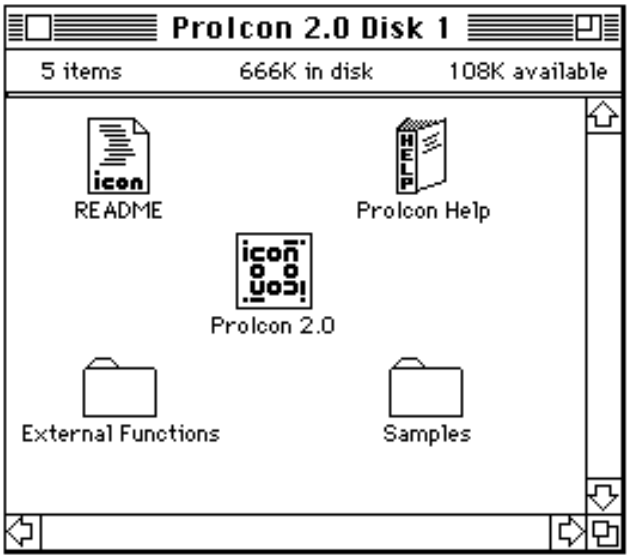
Installing ProIcon

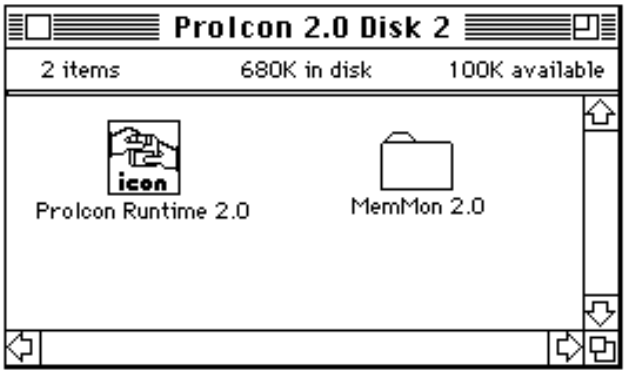
Configuration Requirements

ProIcon requires System Version 6.0.1 or higher and at least 400 KB of free memory; we recommend 1MB RAM. The memory-monitoring application, MemMon, requires an 8-bit color or gray-scale monitor. MemMon is not needed to write or run ProIcon programs.

Installing ProIcon is easy. First decide where you want it to reside. We suggest that you create a folder named ProIcon just for that purpose. When you've done that, copy the contents of the two ProIcon 2.0 disks to this folder. The disks look like this:

ProIcon Disks





If you're short of space, you may not want to include everything. All that's absolutely necessary is the ProIcon application itself. However, if you don't include ProIcon Help (which must be in the same folder as the ProIcon application or in your System folder), you won't be able to use online help. If you don't include ProIcon Runtime 2.0, you won't be able to run previously compiled Icon programs directly but instead will have to run them from the ProIcon 2.0 application.

README is a text file that contains last-minute notes about things that didn't get into this manual. You can open README with any text editor or ProIcon. Be sure to read this file; it may contain important information. You may wish to print a copy to keep with your ProIcon documentation. There also may be important README files in folders.

The Samples folder on the first disk contains sample Icon programs and some interesting data. The External Fucntions folder on that disk contains HyperCard XCMD and XFCN material as well as samples for ProIcon 2.0's external function interface (see Appendix D).

The MemMon 2.0 folder on the second disk contains a separate application for viewing memory management in ProIcon (see Appendix E). There also are sample files.

A Guide to the Manual

This manual is divided into four parts:

- The introduction that you're reading.
- A User's Manual that describes the ProIcon application for the Macintosh.
- Supplementary material on Icon, which goes beyond the basic reference: *The Icon Programming Language*.
- Appendices and an index.

The remainder of this part tells you more about ProIcon in a general way, including its relationship to the Icon programming language and a special introduction to Icon for readers familiar with the SNOBOL4 programming language.

The User's Manual begins with a quick tour of ProIcon, then introduces you to compiling and running programs, the online help facility, entering and editing text, and menu reference.

Part three contains a guide to the most current version of the Icon programming language and special features of ProIcon, as well as a mini reference manual.

The appendices cover character codes, international string comparison, Icon language checklists, external functions, memory monitoring, and references.

2

More About ProIcon

More About ProIcon

If You're New to Icon ...

Icon is an exciting, high-level, general-purpose programming language that has unusual features like high-level operations on strings and structures, pattern matching, goal-directed evaluation, dynamic typing, and run-time creation of data structures. Icon's rich set of operations and control structures and its powerful expression-evaluation mechanism give you the ability to write concise solutions to complex problems easily and quickly.

You can use Icon for rapid prototyping of large systems or for quick reformatting of data, for problems in artificial intelligence, for text analysis, or for quickly changing complex files.

Icon was developed at The University of Arizona. It's the most recent language resulting from a long line of research that produced languages like SNOBOL4. On the surface, Icon looks a lot like many other modern programming languages. It has a syntax similar to that of C and Pascal, and it has many of the traditional control structures you've come to expect, but Icon is much more powerful. It has generators, expressions that may produce more than one result, and it has goal-directed evaluation, which allows many complex computations to be expressed in concise and natural ways — searching for solutions is done automatically.

Icon is a rich language. It has an extensive repertoire of operations on strings of characters and supports a variety of sophisticated data structures. The way Icon evaluates expressions allows many programming tasks to be expressed more

naturally and concisely than they can be in most other programming languages.

This richness makes it easy to write programs that perform complicated tasks. Consequently, there's a lot you need to know in order to learn to use Icon effectively. Fortunately, you can get started with just a small part of the entire language. Then, to use Icon's capabilities to their fullest, you'll want to learn more. Learning Icon can be fun, too. Icon has lots of new ideas and is full of exciting features. You'll not be bored.

A few words about strings and structures: As everyone knows, the first computers were designed to solve numerical problems — the computation of trajectories, orbits, and so forth. It wasn't long before the potential of computers to manipulate symbolic data like equations and natural language was recognized. Arranging data in more complex ways like trees and lists followed — and a whole new world of computation was opened.

While Icon has the usual capabilities for numerical computation, its real strengths lie in the manipulation of symbolic data and structures. In Icon, symbolic data is represented by strings (sequences) of characters. Virtually every kind of data can be represented by strings — everything from the text of a manual like this to mathematical formulas. Text files on computers are just strings. Icon's repertoire for string processing provides many ways to analyze and synthesize strings. Analysis operations make it easy to find patterns in text, things like words and parenthesized expressions. Synthesis operations make it easy to build up complex strings out of simpler ones and to format them in any desired way.

The key to managing complex data is organization. Structures provide this facility. Different situations and different kinds of data need different kinds of organizations. Icon offers several kinds of structures for diverse needs. Lists allow values of any kind (not just characters) to be arranged in sequence and accessed by position. Lists also can be accessed like stacks and queues for programs that need to handle data in these ways. Icon also provides sets, which are unordered collections of values. Operations on sets like union and intersection provide easy ways to handle problems that deal with common attributes and properties. Icon has

more, including tables that provide associative lookup using keys of any type. Icon also lets you build up complicated structures of your own. There is no end to the possibilities.

Many real-world problems have more than one solution. Sometimes it's necessary to select among those solutions to satisfy a constraint or to get a solution to a larger problem. Icon provides a very powerful method for doing this. Expressions, called generators, can produce sequences of values. Goal-directed evaluation causes generators to produce results to reach combinations that satisfy larger requirements.

In short, Icon is a powerful, high-level programming language. The things it allows you to do quickly and easily make programming more fun and less of a chore. And being able to do complex things so easily may encourage you to try tasks you've always wanted to do but couldn't face in other programming languages. Icon may well expand your programming horizons.

Icon for SNOBOL4 Programmers

If you're a SNOBOL4 programmer, you'll find that Icon has many of the familiar features of SNOBOL4. Some of these features are cast in somewhat different ways in Icon. Since Icon evolved from SNOBOL4, it has a lot of features that SNOBOL4 doesn't. It also lacks some of the more esoteric features of SNOBOL4. The main differences between the two languages are described in the following sections.

Syntax

As you probably know, SNOBOL4 has a syntax all its own. It has several different statement types that focus on a subject on which operations are performed, such as assignment and pattern matching. SNOBOL4 has no control structures for expressing looping or selection; it relies on conditional operations and gotos to control the sequence of execution. Although this mechanism is very general, the need to fabricate loops, for example, makes programming tedious. It's also difficult to write SNOBOL4 programs that are 'well structured' — programs that are easy to understand and modify and that are free from sneaky bugs in control flow. SNOBOL4 programs also are monolithic. Despite the facility to define functions, all the statements are really part of the whole program.

An Icon program, by contrast, consists of a set of modules — procedures that are logically and physically distinct from each other. Thus, a program to be divided into its logical components and even be kept in a number of different files.

An Icon procedure consists of expressions. Expressions, unlike statements, are made up of other expressions; there are no sharp demarcations as there are with statements. Icon has several control structures that allow common forms of expression evaluation to be cast in standard, understandable, and well-structured ways. Icon doesn't have labels and gotos. It doesn't need them.

If you are familiar with a language like Pascal or C, you'll probably not have much trouble phrasing your programs in

terms of Icon's syntax. Otherwise, you'll need to spend some time learning how to write programs using control structures without labels and gotos. Copying a file provides a comparison. In SNOBOL4 it is

```
read        OUTPUT = INPUT     :S(read)
```

while in Icon, it is

```
while write(read())
```

Both are short and the logic is the same; they just look different.

Success and Failure

As the preceding example suggests, Icon uses the SNOBOL4 concepts of success and failure to control program flow. In this, you have an advantage over a person who knows only programming languages like BASIC and Pascal, and for whom the idea of failure may be completely new. In Icon, as in SNOBOL4, the failure of an operation prevents evaluation of other surrounding operations. While failure is used to select conditional gotos in SNOBOL4, failure is used to drive control structures in Icon. For example, a conditional assignment in SNOBOL4 might look like this:

```
            i = LT(i,j) j                    :S(next)
            i = 0
next
```

In Icon, it looks like this

```
if i < j then i := j else i := 0
```

Once you get used to it, you'll see how similar the use of success and failure is in SNOBOL4 and Icon. In fact, as the example above illustrates, the control structures of Icon just use implicit gotos and labels — you don't have to write them yourself as you do in SNOBOL4.

Pattern Matching

One of the most distinctive features of SNOBOL4 is pattern matching and its repertoire of built-in patterns from which you can build more complex ones of your own.

Patterns provide a high-level of abstraction that describes the structure of strings without, for the most part, the need to specify how matching is actually done. In the strengths of patterns lie their weaknesses. It's difficult to specify the matching process if you need to, and most of the rest of the facilities of SNOBOL4 are not available in pattern matching except by using awkward and contrived constructions.

Icon approaches the problem of pattern matching in a somewhat different way. Instead of having patterns, Icon has matching functions. Icon focuses more on the ability to express how the matching is done than it does on the description of what is matched. This allows all the features of Icon to be used in pattern matching, but at the expense the high-level of abstraction that SNOBOL4 patterns allow.

For example, writing out the comma-terminated substrings of a string in SNOBOL4 might be done this way:

```
                P = BREAK(",") . OUTPUT LEN(1)
next            S ? P =                 :S(next)
```

In Icon, it might look like this:

```
s ? while write(tab(upto(',')))
  do move(1)
```

The point is that in SNOBOL4 you think of a pattern that matches the desired string, while in Icon you specify how to match the desired string. Granted, the two methods look similar; it's the viewpoint that's different. It's worth noting that Icon allows any operation in the matching process, while SNOBOL4 doesn't. In SNOBOL4, for example, you can't just make an assignment to OUTPUT; you have to use a different operation that associates the name OUTPUT with a pattern.

Actually it's possible to capture SNOBOL4's concept of patterns in terms of Icon procedures, although the method for doing so requires discipline and adds overhead in execution time. For example, you could write an Icon procedure that does the matching shown above:

```
procedure p()
  while write(tab(upto(','))) do
    move(1)
end
```

and use it as if it were a pattern:

```
s ? p()
```

A word of advice here: Trying to write pattern matching in Icon the way you would in SNOBOL4 often leads to frustration and awkward programs. It's much better to spend the time getting used to the way Icon does pattern matching and to use Icon for its strengths rather than to try to force it to do things the way SNOBOL4 does.

Scope

Scope is a technical term that refers to the portion of a program in which an identifier is known and in which its value is accessible. At first glance, you may not see the differences in scoping of identifiers in SNOBOL4 and Icon. There are significant differences, however.

In SNOBOL4, every identifier is known throughout the entire program. There is no distinction between global and local identifiers. If an identifier is an argument or is "local" to a defined function, its value is saved when the function is called, a new value is assigned, and the old value is restored when the function returns. The current value of the identifier, however, is available to the entire program. This is a form of "dynamic scoping".

Icon, on the other hand, has two kinds of identifiers: global and local. Global identifiers are available to all the procedures in the program. Local identifiers, however, come into existence when a procedure is called, are accessible only within that call of the procedure, and are destroyed when the procedure returns. This kind of scoping is called "static".

There are advantages and disadvantages to both kinds of scoping. Dynamic scoping makes it trivially easy to associate identifiers with patterns, as in

```
BREAK(",") . OUTPUT
```

and allows such patterns to be used throughout the program. You can't do this in Icon, since local identifiers come and go and there may be several with the same name. However, there's no way for a defined function in SNOBOL4 to have an identifier all its own that no other function can change. Icon's scoping is more conventional than SNOBOL4's. More than likely, you'll not notice the difference, although

with Icon's scoping you're less likely to have obscure bugs that result from two procedures accidentally sharing the same identifier.

Data Structures

Some of Icon's data structures are very similar to those of SNOBOL4. For example, Icon has tables that are nearly identical to those of SNOBOL4; you should have no trouble using Icon's tables if you are familiar with those of SNOBOL4. Icon's lists are similar to SNOBOL4's arrays, although Icon's lists have only one dimension. Icon's lists, however, can grow and shrink and be used as stacks and queues. Icon also has sets, which are just collections of values — very handy for cases where you want to keep a collection of things together. For example, to write out all the different words in a file using Icon, the following will do:

```
words := set()
every insert(words,nextword(f))
every write(!sort(words))
```

Here `nextword(f)` is a procedure that generates the words from the file `f`. The last line of this program segment may look mysterious. It hints at the power of Icon.

What's New with Icon

In addition to the features mentioned above, Icon has many features SNOBOL4 doesn't have. Perhaps the most exciting and useful feature is the ability of an Icon expression to generate a sequence of values. While a SNOBOL4 expression can only fail or succeed, an Icon expression may fail, produce a single result (succeed), or generate many results. Generation is particularly useful for operations that naturally have several results, such as the position at which one string occurs as a substring of another. (You may sense the germ of this idea in the few SNOBOL4 patterns that can match in more than one way.)

Generators lead to all kinds of interesting possibilities and give Icon's expression evaluation a two-dimensional character, as opposed to programming languages in which every expression produces exactly one result. It's here that you'll find the most interesting aspects of Icon and discover how they can be used to cast complex operations in natural and

concise ways. Here's just a hint. The following example writes all the common positions where s1 is a substring in s2 and s3:

```
every write(find(s1,s2) = find(s1,s3))
```

What's Missing in Icon

Icon lacks some of SNOBOL4's most powerful features — the ability to modify a program during execution, the ability to compile and execute strings on the fly, and the ability to redefine functions and operators dynamically. These features of SNOBOL4, while very powerful in terms of what you can do during program execution, are also expensive in terms of the implementation and the kind of program structure needed to make them feasible.

If you're used to these esoteric features of SNOBOL4, you may be disappointed initially when programming in Icon. In most cases, the same results can be achieved in Icon by using different techniques. For those things that can't be recast in Icon, you may want to continue using SNOBOL4. However, there also are things you can do easily in Icon that have no natural counterpart in SNOBOL4. Most programmers who know both SNOBOL4 and Icon do most of their programming in Icon.

About the Name "Icon"

As a Macintosh user, you may think that using the name "Icon" for a programming language is misleading or possibly a subtle pun. Actually the choice of "Icon" as the name for the programming language you are about to use has nothing to do with the Macintosh. The name was chosen in 1976, before the word came into use to describe the pictograms that identify files and functions on the screen of the Macintosh and other computers.

Naming a programming language is more difficult than you might think. Should the name be an acronym derived from a descriptive phrase (like FORTRAN)? Should it try to connote some salient property of the language (like EASy)? Should it honor some important person (like Ada)? Should it be clever and catchy (like Spitbol)? Or what? There are other considerations. Is the name easy to remember? What kind of images does it evoke? Is it easy to spell? Easy to pronounce? Distinctive? Pleasant?

The name "Icon" for this programming language is not an acronym (you might be able to dream up a suitable phrase, though), and it has no special meaning (but you might well conjure one up). It was chosen to be short, crisp, and uncomplicated. And because the designers had run out of other ideas. Had they foreseen the use of the word "icon" for those little pictograms, they certainly would have picked another name.

Granted, the name for the language may evoke the wrong image and be taken for something it isn't. Unfortunately, a name once chosen, published, and put into use is not easily changed. In the end, such names really mean nothing in themselves and you'll quickly forget the confusion. However, you may find yourself having to explain the problem to your fellow Macintosh users more than once.

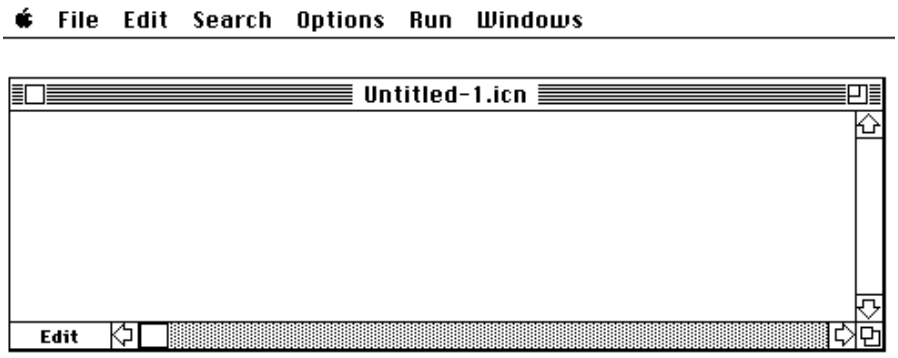
3

A Quick Tour

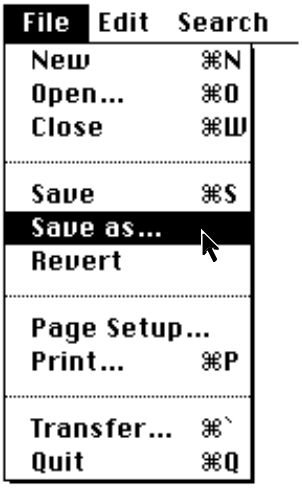
A Quick Tour

In this chapter you'll see how ProIcon works and how to create, run, and correct Icon programs. This is just a start – there's more information in the chapters that follow.

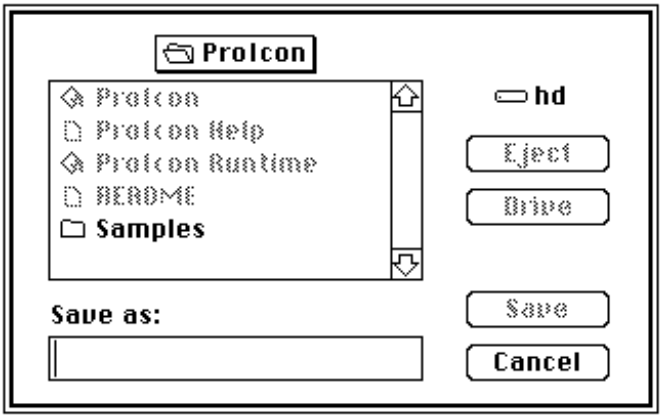
Start ProIcon by double-clicking on its icon. You'll see the ProIcon menu bar and an open window waiting for a new program:



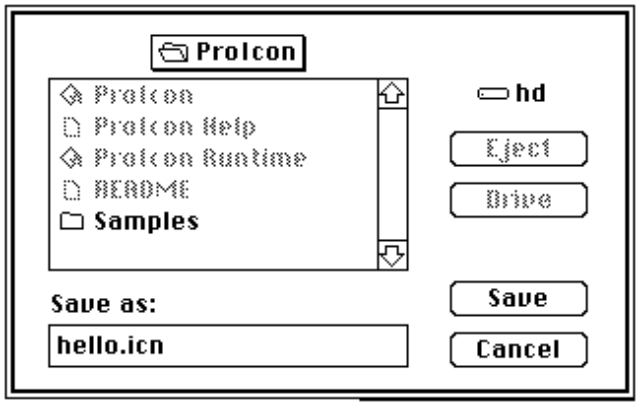
The name Untitled-1.icn identifies a window that hasn't yet been given a name and saved in a file. Before you go any further, it's a good idea to name the window and connect it to a file that will be saved. Pull down the File menu and select Save as ... :



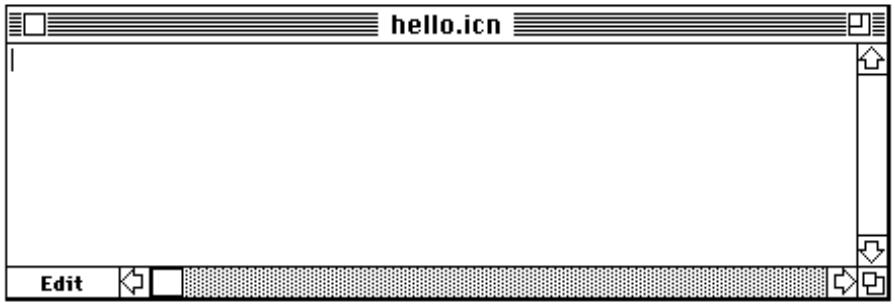
You'll get a file dialog box that lets you navigate to the folder in which you want to save your program and specify its name:



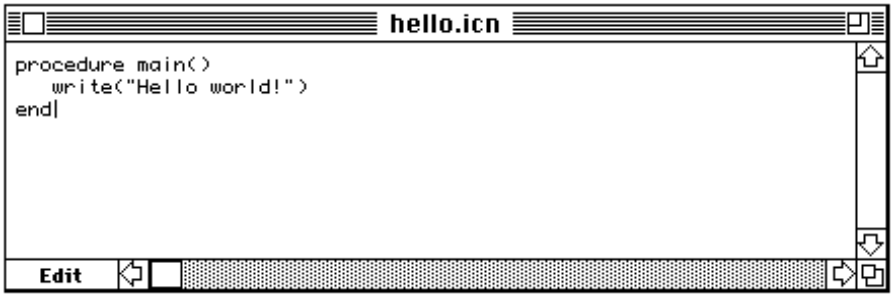
The file name for an Icon program must end in .icn. Suppose you want to name your program hello.icn. Enter hello.icn in the box and select Save.



The title of your program window changes accordingly:



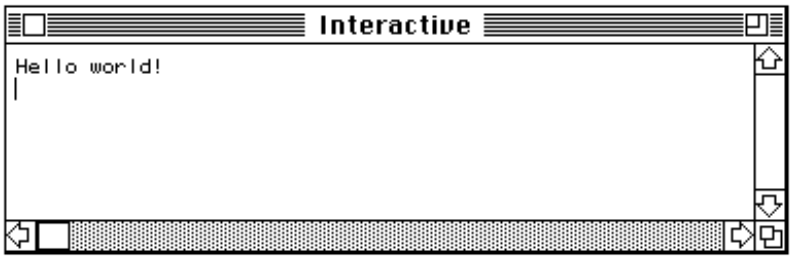
Now you're ready to enter a program. Keep it simple to start with, such as:



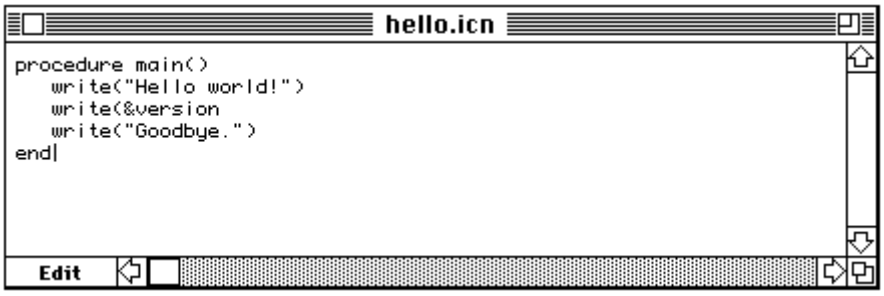
When you enter this program, you're using the ProIcon editor. It's a program editor, so you type return at the end of every line of the program. ProIcon's editor follows the Apple Human Interface guidelines. If you're familiar with other Macintosh editors, ProIcon's editor should seem very natural to you. Chapter 6 contains detailed information about the ProIcon editor. For now, just do what comes naturally.

Now you're ready to run your first ProIcon program. Pull down the Run menu and select Compile Window:

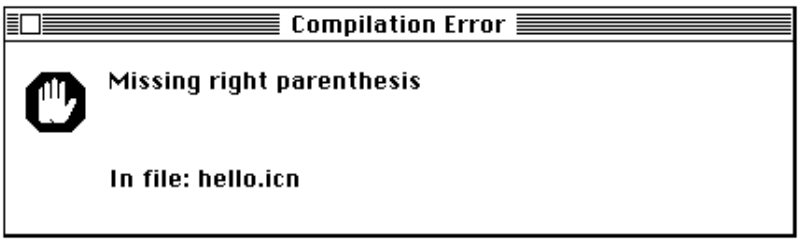
The cursor changes to a "spinning beachball" while ProIcon compiles and links your program. Your program then executes and the output pops up in a window named Interactive:



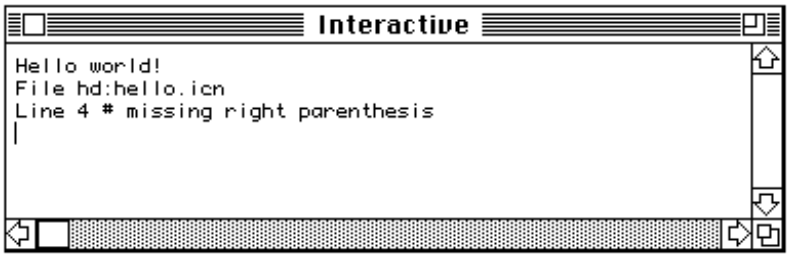
Now you can change your program. Suppose you want to know what version of ProIcon you're running. Add two lines so that you have



To run this program, use the ⌘-K shortcut that you saw in the Run menu. Oops – there was a mistake in the program:



The error and location also are noted in the Interactive window following the previous output:



The compiler finds the error in line 4 when it's looking for a parenthesis to complete line 3 and does not find it. Close the window named Compilation Error to get rid of it. Notice

that the cursor is positioned at the line of your program window where the compiler found the error. Generally speaking, the line where an error is detected is not exactly where you made the mistake; the compiler can't find an error until something actually syntactically incorrect occurs (you could have had the missing parenthesis at the beginning of line 4, for example).

Correct your program and run it again to be sure it's right. When you're finished, save your program, if you want, by selecting Save from the File menu. If you close the program window without saving changes, you'll be asked if you want to save your modified program.

When you're finished with a ProIcon session, leave the application by selecting Quit from the File menu (or use the ⌘-Q keyboard shortcut).

That's the basic idea. Enter a program (or bring in an existing one using Open... from the File menu), run it, make corrections and re-run it as necessary, and save your program when you're finished.

Of course, there's a lot more to ProIcon. The next chapter describes more features of ProIcon.

Read on!

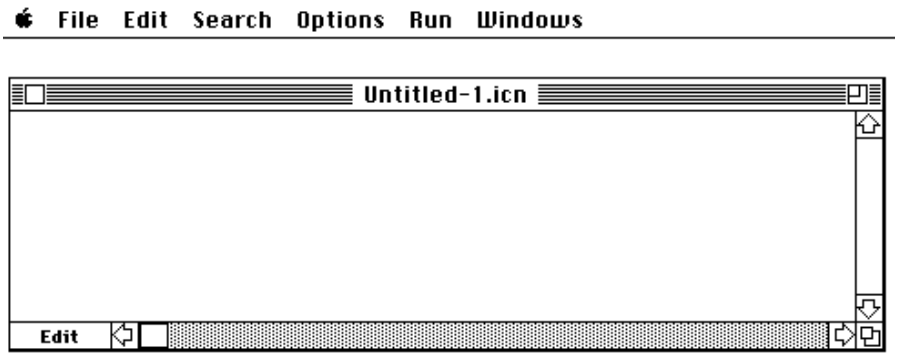
3

A Quick Tour

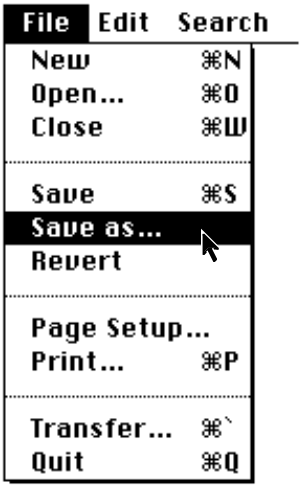
A Quick Tour

In this chapter you'll see how ProIcon works and how to create, run, and correct Icon programs. This is just a start – there's more information in the chapters that follow.

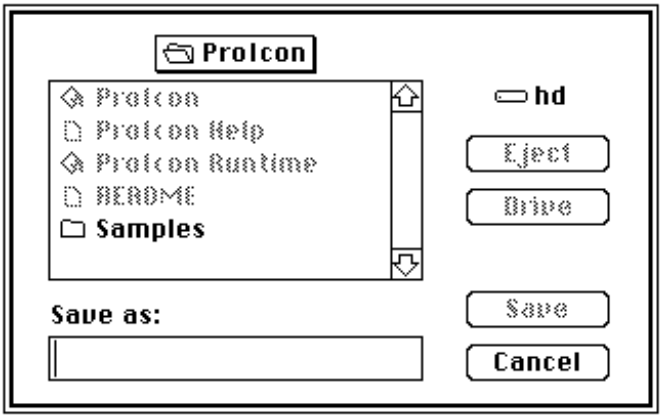
Start ProIcon by double-clicking on its icon. You'll see the ProIcon menu bar and an open window waiting for a new program:



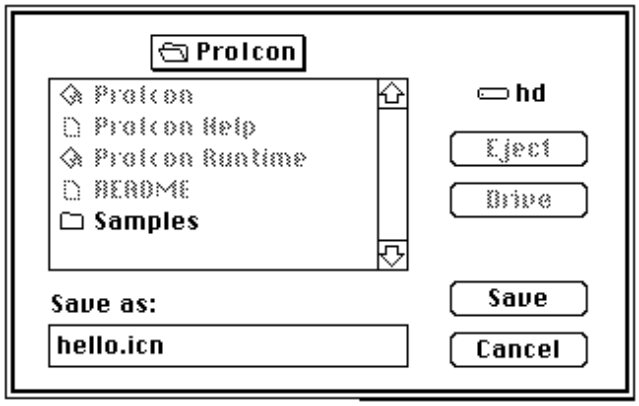
The name Untitled-1.icn identifies a window that hasn't yet been given a name and saved in a file. Before you go any further, it's a good idea to name the window and connect it to a file that will be saved. Pull down the File menu and select Save as ... :



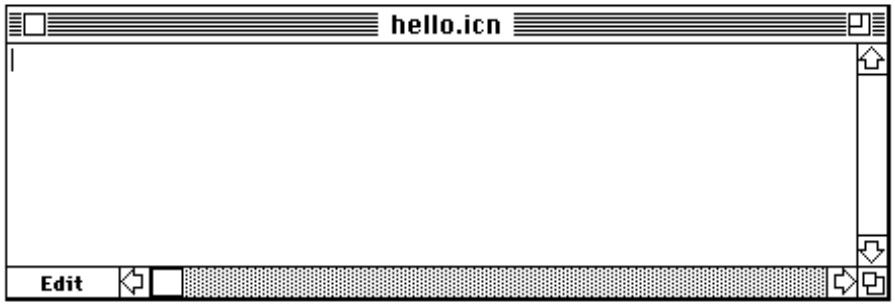
You'll get a file dialog box that lets you navigate to the folder in which you want to save your program and specify its name:



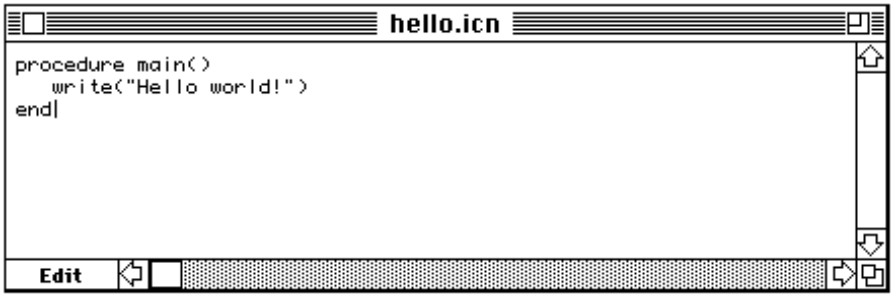
The file name for an Icon program must end in .icn. Suppose you want to name your program hello.icn. Enter hello.icn in the box and select Save.



The title of your program window changes accordingly:



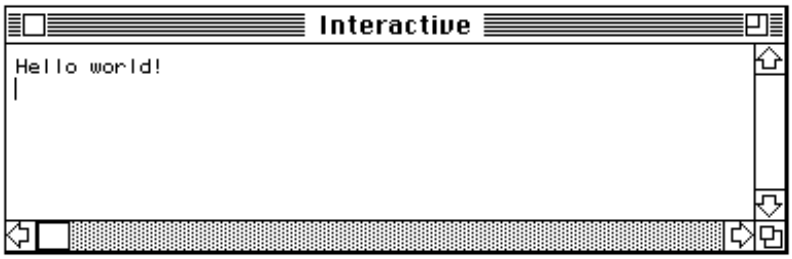
Now you're ready to enter a program. Keep it simple to start with, such as:



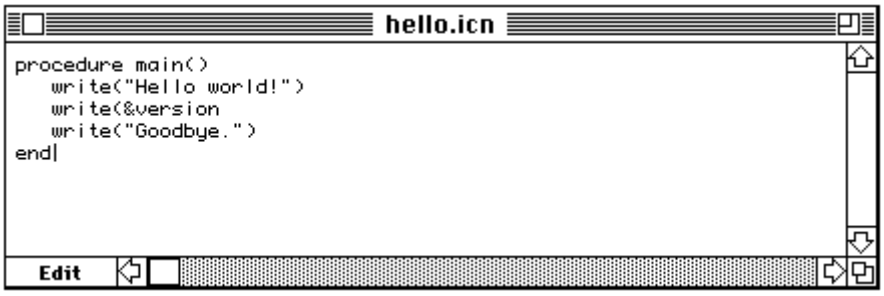
When you enter this program, you're using the ProIcon editor. It's a program editor, so you type return at the end of every line of the program. ProIcon's editor follows the Apple Human Interface guidelines. If you're familiar with other Macintosh editors, ProIcon's editor should seem very natural to you. Chapter 6 contains detailed information about the ProIcon editor. For now, just do what comes naturally.

Now you're ready to run your first ProIcon program. Pull down the Run menu and select Compile Window:

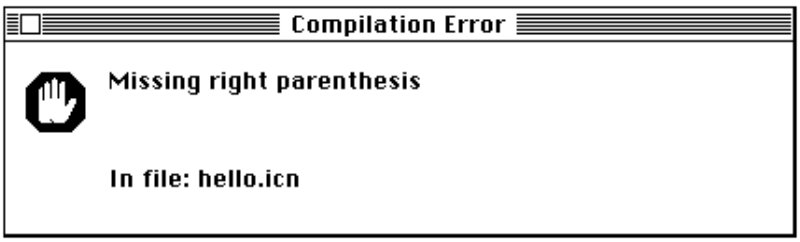
The cursor changes to a "spinning beachball" while ProIcon compiles and links your program. Your program then executes and the output pops up in a window named Interactive:



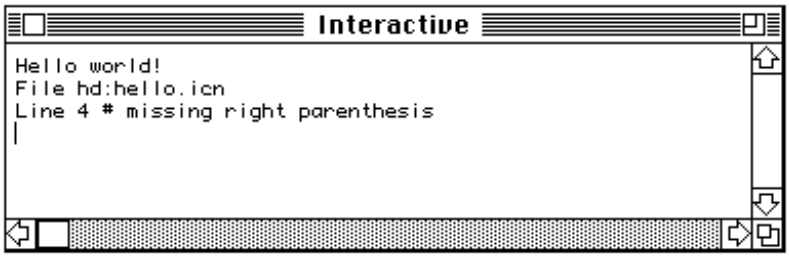
Now you can change your program. Suppose you want to know what version of ProIcon you're running. Add two lines so that you have



To run this program, use the ⌘-K shortcut that you saw in the Run menu. Oops – there was a mistake in the program:



The error and location also are noted in the Interactive window following the previous output:



The compiler finds the error in line 4 when it's looking for a parenthesis to complete line 3 and does not find it. Close the window named Compilation Error to get rid of it. Notice

that the cursor is positioned at the line of your program window where the compiler found the error. Generally speaking, the line where an error is detected is not exactly where you made the mistake; the compiler can't find an error until something actually syntactically incorrect occurs (you could have had the missing parenthesis at the beginning of line 4, for example).

Correct your program and run it again to be sure it's right. When you're finished, save your program, if you want, by selecting Save from the File menu. If you close the program window without saving changes, you'll be asked if you want to save your modified program.

When you're finished with a ProIcon session, leave the application by selecting Quit from the File menu (or use the ⌘-Q keyboard shortcut).

That's the basic idea. Enter a program (or bring in an existing one using Open... from the File menu), run it, make corrections and re-run it as necessary, and save your program when you're finished.

Of course, there's a lot more to ProIcon. The next chapter describes more features of ProIcon.

Read on!

Compiling and Running Programs

In the last chapter you saw the basic ideas for compiling, running, and modifying ProIcon programs. ProIcon is a powerful tool that provides many facilities to help you build applications written in ProIcon. You'll want to use some of these facilities as a regular part of your work with ProIcon. You may need other facilities only in special situations. This chapter starts with a description of facilities that you will probably use a lot. Facilities for special situations are near the end of this chapter.

Compilation and Linking

When you compile a ProIcon program (using, for example ⌘-K), three processes normally take place: first compilation proper, then linking, and finally execution. The compilation phase converts your program into an intermediate form. Linking converts the intermediate files to an executable file. The executable file is then run.

You'll notice the Run menu has two options that are checked by default: Link after Compile and Run after Link. As long as these options are checked, all three processes take place, one after another. You can, however, stop after any process. For example, if you uncheck Link after Compile (by clicking on it in the Run menu), ProIcon stops after compiling your program and you get two intermediate files as a result (they are deleted if you link after compiling). These intermediate files have the suffixes .u1 and .u2 in place of the suffix .icn in the name of your

program.

Libraries

You might ask: "What good are intermediate files?" The answer to this question is suggested by the linking process mentioned above. The ProIcon linker can combine intermediate files from several programs to produce a single executable file. This is useful for making libraries of procedures that can be included in many programs.

If you want to include intermediate files from a library program, you must have a link declaration that names the library in your program. For example, suppose you have a collection of procedures for doing rational arithmetic and you place them in a file rational.icn. Compile this file but don't link it (as described above). Now suppose you want to include this library in a program in approx.icn. This file should start as follows:

```
link rational

procedure main()
   . . .
```

When you compile and link approx.icn, the intermediate files from rational.icn are included by the linker, so you can use any procedure contained in rational.icn within approx.icn. You'll notice that just the name rational is used in the link declaration (no suffix). And, of course, rational.icn must not contain a main procedure, since only one main procedure is allowed in a program and a program that includes library procedures normally provides it. You can link several libraries, either with separate link declarations or by using a comma-separated list, as in

```
link rational, input, output
```

If the name of a library does not satisfy the syntactic requirements for an Icon identifier, it must be enclosed in quotation marks, as in

```
link "long-arith"
```

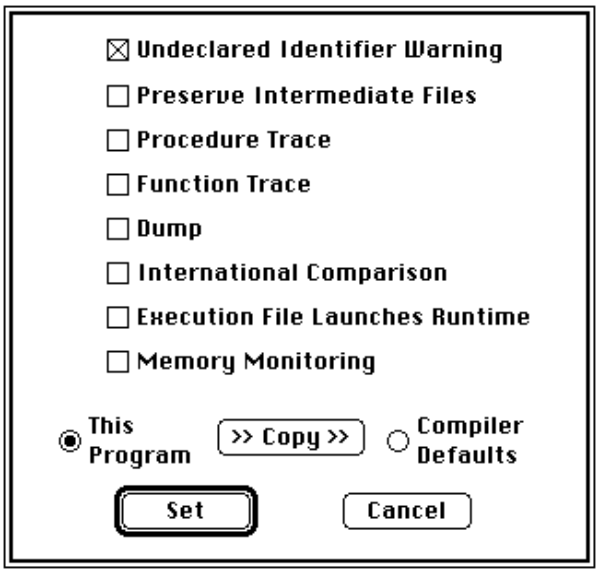
You can also place intermediate files for library procedures in different folders and arrange for ProIcon to find

them automatically without having to specify the paths. The way to do this is discussed later in this chapter.

Icon Applications

Executable files produced by the linker have names corresponding to the Icon program files from which they are produced, but without the .icn suffix. For example, the executable file that results from compiling and linking hello.icn is named hello. You can run an executable file without having to recompile the program. Select Run File ... from the Run menu. You get a file dialog showing the available executable files. When you select one and complete the dialog, it runs.

So far, you've been working inside the ProIcon application. You don't have to be in the ProIcon application to run an executable file. Just launch it from the desktop. This lets you build applications in ProIcon and run them as you wish. You can also give such applications to others. It is necessary to include the file ProIcon Runtime because it's needed to support executable files. It's another file to remember, but there's a good side to it — all executable files use the same ProIcon Runtime and hence executable files are small.



Program Options

Selecting Program Options ... from the Options menu lets you turn some program options on and off:

Undeclared Identifiers

Icon does not require you to declare local identifiers. If an undeclared identifier appears in a procedure and there is no global declaration for it anywhere in your program, the identifier is taken to be local. This saves having to write a lot of declarations. It can be dangerous, however. You may intend an identifier to be local, but there may be a global declaration for it that you don't notice. If you check Undeclared Identifier Warning, ProIcon's linker issues a warning message if it finds an undeclared identifier. Such messages are just warnings; they don't prevent your program from linking or running.

Intermediate Files

Normally, when ProIcon compiles and links a program, it deletes the intermediate files produced by the compiler (the .u1 and .u2 files). If you want to keep these files (to use in a library as described earlier in this chapter, for example) check Preserve Intermediate Files.

Tracing

ProIcon offers two kinds of tracing: procedure tracing and function tracing. Check the corresponding boxes if you want tracing to be done automatically. A word of caution: tracing produces a lot of output and it may be

hard to find what you want. Function tracing produces much more output than procedure tracing in most programs. While checking these options gives you tracing without having to modify your program, you may find it more useful to add code to your program to turn tracing on and off selectively by setting the values of &trace and &ftrace.

Termination Dumps

If you check Dump, you'll get a listing of variables and their values when your program terminates. Again, this also can be done by setting &dump in your program.

String Comparison

If International Comparison is checked, ProIcon compares strings using the Macintosh international comparison system. See Appendix B for details.

Memory Monitoring

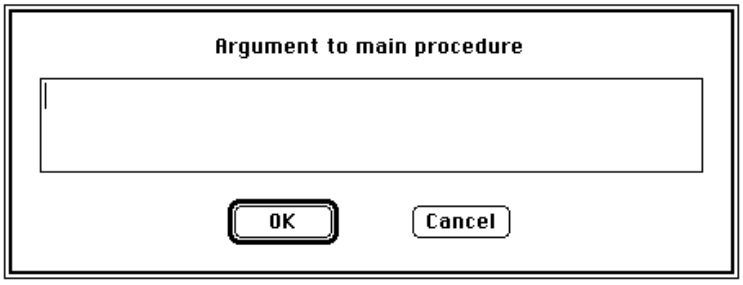
Memory Monitoring causes information about storage allocation and garbage collection to be recorded. Checking this box brings up an Open dialog when you run the program for a file to hold the information. Appendix E describes how to use the results.

Launching Runtime

Object File Launches Runtime determines what happens when you launch an executable file from the desktop. If this box is not checked, launching an executable file starts the ProIcon application. If the box is checked, only the ProIcon run-time system is activated. You probably want the ProIcon application to be launched for your own programs, but you'll need to have the run-time system launched for programs you give to others, since they may not have the ProIcon application.

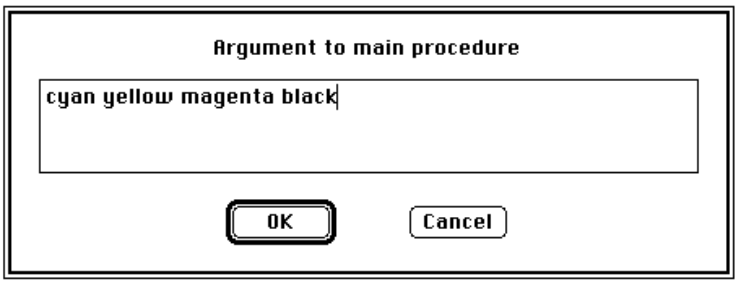
Compiler Defaults

As indicated by the radio button in the Program Options box, the options you specify normally just apply to the current program. If you want them to apply to all programs, click on the Compiler Defaults radio button. You also can copy from one to the other by clicking on the Copy button.



Parameter Strings

Parameters can be communicated to a ProIcon program by means of a blank-separated string that provides a list as the argument to the main procedure. This string is entered



using the Parameter String ... entry from the Options menu:

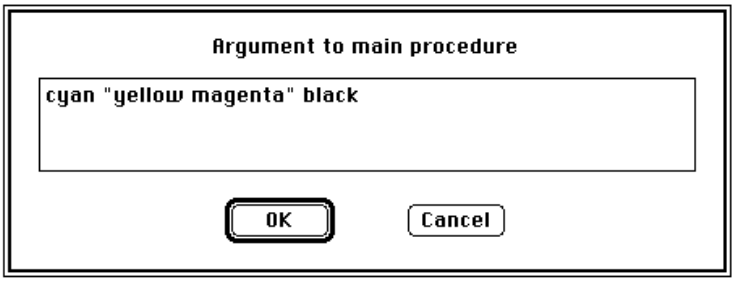
For example, if you enter the string

```
cyan yellow magenta black
```

giving

then in

```
procedure main(args)
   every write(!args)
end
```



the values written are

```
cyan
yellow
magenta
```

```
black
```

You can include blanks in an argument by enclosing the argument in quotation marks, as in

for which the values written are

```
cyan
yellow magenta
black
```

Path Names

Full Path Names

For simple file names in link declarations, ProIcon searches the folder where your program resides. If you want to look in other folders, you may specify files by full or partial path names.

Partial Path Names

A *full path* contains several elements, each separated by a colon. A full path name begins with a disk name, then zero or more folder names, and ends with the file name. Two examples are:

```
link "hd:ProIcon:Library:rational"
link "Floppy Backup:ProIcon:test
```

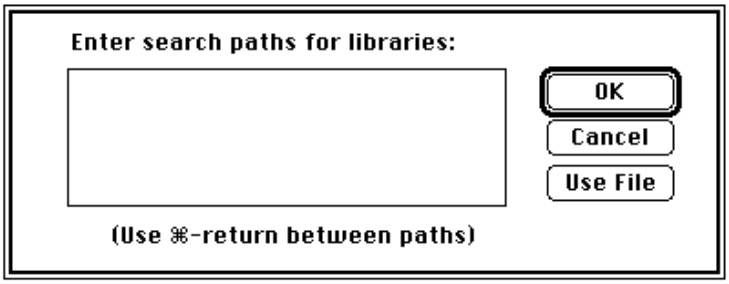
A *partial* path is relative to the current working folder, and begins with a colon, as in:

```
link ":Library:rational"
link ":ProIcon:test"
```

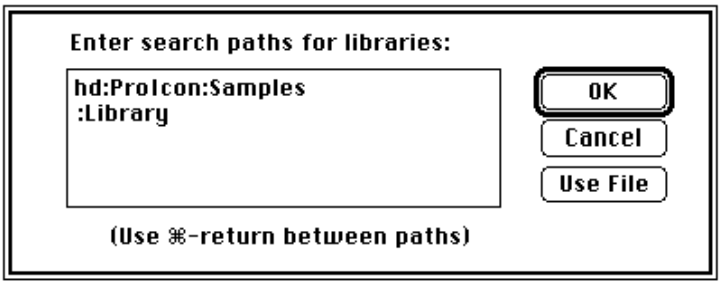
In both full and partial path names, two adjacent colons mean “go up one folder” in your disk’s folder hierarchy.

Library Folders

If you use libraries of intermediate files, you’ll probably find it convenient to keep them in one or more folders, separate from the rest of your ProIcon work. While you can specify the paths to such folders in link declarations, you’ll probably find it more convenient to use ProIcon’s library search facility.



When ProIcon's linker encounters a link declaration for an intermediate file without a full path name, it searches



for the file first in the folder that contains your main program. If the linker doesn't find the file there, it uses paths specified in the Library Folders ... selection in the Options menu. The specification of paths in the previous section goes to the file level. Here it goes to the folder level. To set up library search paths for folders, select Library Folders ... :

As indicated, you can specify one or more path names, separated by ⌘-returns. For example:

Once these paths are entered, they are used when linking all your ProIcon programs, even if you quit the application and then launch it again. You can, of course, change the paths any time you wish.

Wildcards

Rather than specifying individual folders, your search paths may include classes of names. This is done by using "wildcard" characters.

The ? wildcard character matches any single character in a name. Thus, A?C matches all three-character names whose first letter is A and whose last letter is C, such as

Path Files

AKC or a9c.

The * wildcard character matches zero or more characters in a name. For example, A*M matches AM or Alarm.

If *, ? or \ appear literally in a name, precede each with a backslash, as in *, \?, or \\.

Here are some folder names as they might appear in the search path dialog:

```
Library*
R??d:Library
```

If you use complicated search paths, or change them from time to time, you can record them in text files. Open a new window in ProIcon, and type in the paths, one per line. Save the file with a name that ends in .paths, such as search.paths.

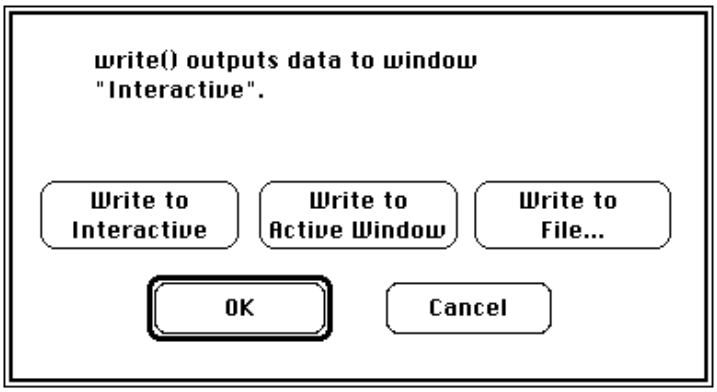
The Library Folders ... dialog includes a Use File button. When you press this button, you can specify the name of the file containing your pre-specified search paths. ProIcon reads in the paths and uses them as if you'd typed them in directly. The .paths suffix is mandatory.

Input and Output

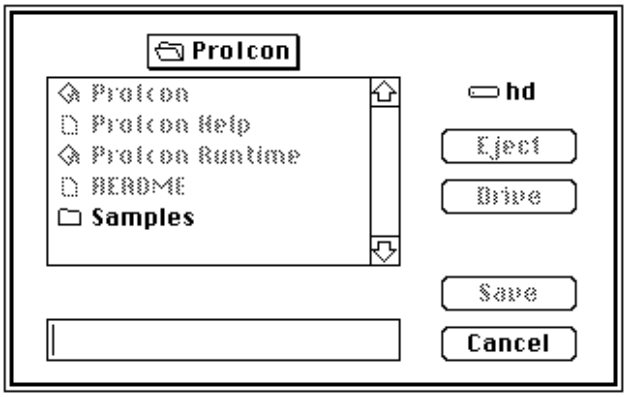
You can open any file you want from inside a running ProIcon program. Three files that you don't have to open are provided for your convenience: standard input, standard output, and error output.

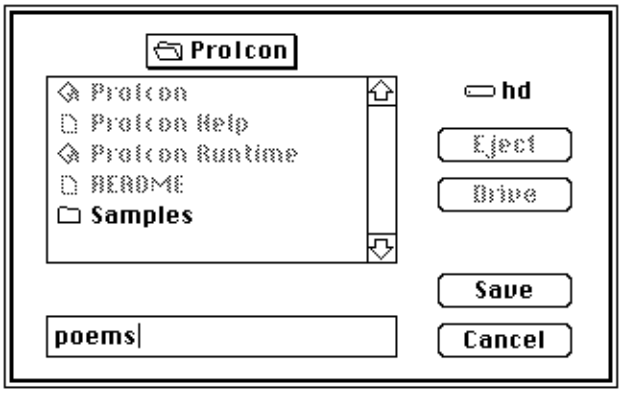
Standard input is where input comes from if you don't specify a file when reading. Standard output is where output goes if you don't specify a file when writing. For example,

```
while write(read())
```



copies standard input to standard output. Error output is





used for error messages. These three files can be specified explicitly in a program as &input, &output, and &errout.

Keyboard End-of-File

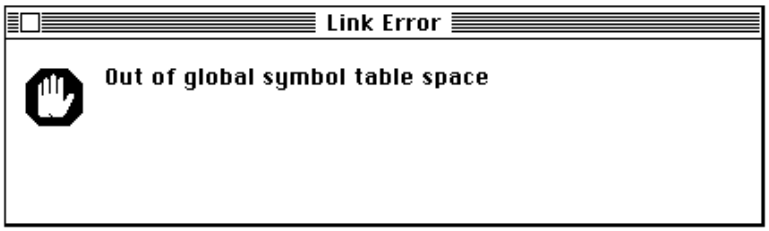
Unless you specify otherwise, standard input is taken from the keyboard, while standard output and error output go to the Interactive window. You can change input or output to the currently active window or to a file. If you select a file that is also opened as a window, your output goes to the window.

Compiler Memory

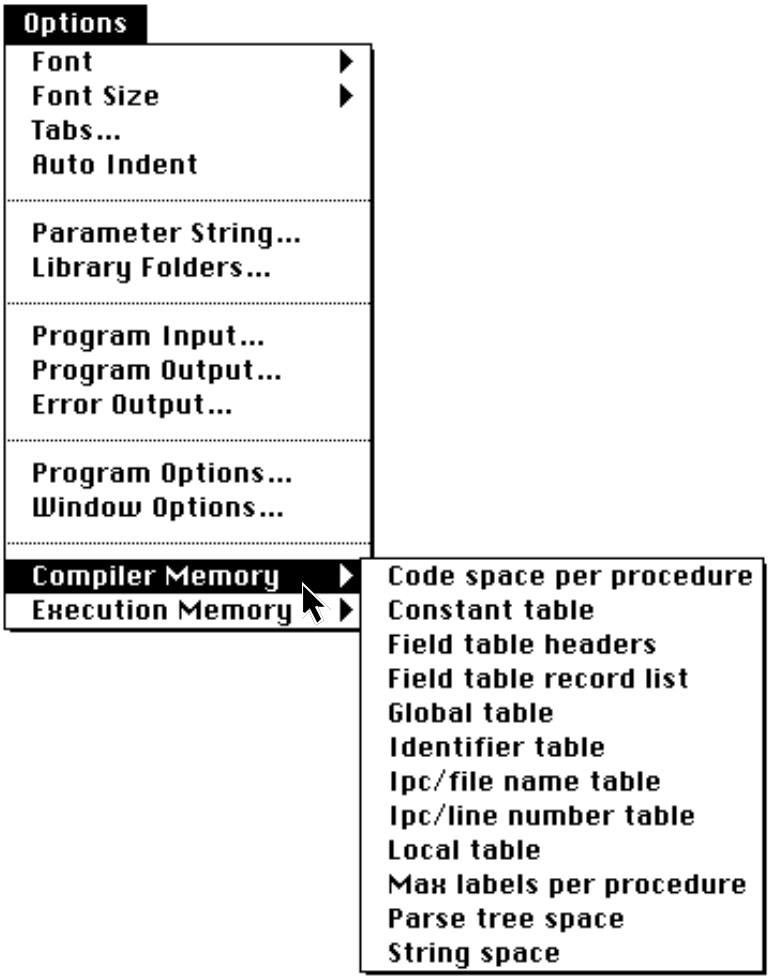
If you want to change program input or output, select Program Input ..., Program Output ..., or Error Output ... from the Options menu. Suppose you want standard output to go to poems. Select Program Output ... :

Click on Write to File... . You get a file dialog:

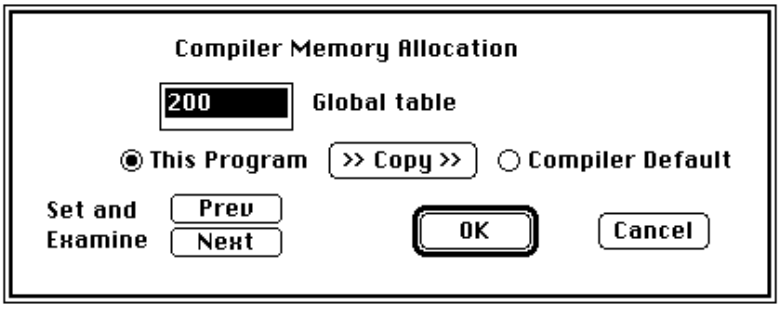
Now enter poems:



Now click OK. If you already had a file named poems, you'd get an alert box that asks you if you want to overwrite it. Complete the dialog as usual and standard output now goes to poems. You can change standard input or error output in a similar fashion.



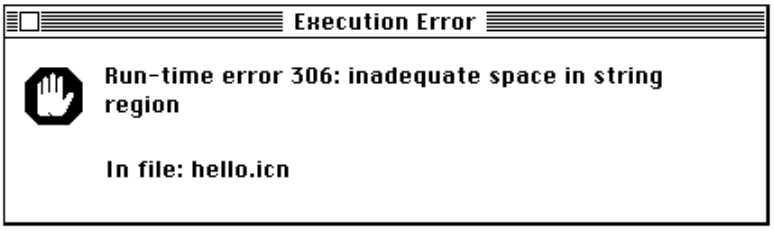
You can terminate keyboard input to a running ProIcon program by selecting Terminate Input from the Edit menu



(or by using the ⌘–D keyboard shortcut).

Memory Management

The ProIcon compiler uses several tables for internal computations. The default sizes for these tables are adequate for most programs, but if you have a very large program or one with unusual characteristics, the compiler may run out of space in one of its tables. If this happens, you get an error message such as



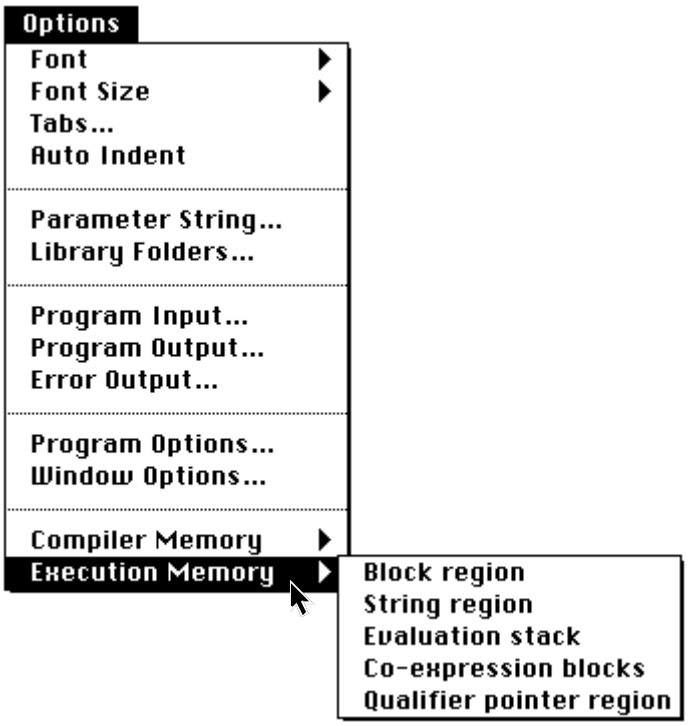
This message means that you have more global identifiers than fit in the compiler's global table. While you might be able to reorganize your program to work around this limit, you also can make the table larger. Select Compiler Memory from the Options menu and pull down the hierarchical menu it points to:

Execution Memory

As you see, there are a lot of sizes you can change. Select Global table as indicated by the error message:

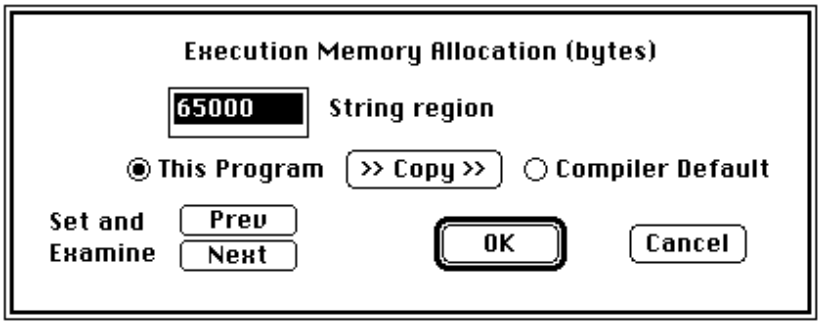
The current size of the global table is highlighted. You can edit this text to change the size. The size to use depends on your program and the specific table involved. You could count the number of global identifiers in your program, but you might just as well double the current value, which probably will be enough with some to spare.

As indicated by the radio button in this box, the size you specify normally just applies to the current program. If you want it to apply to all programs, click on the Compiler Default radio button. You also can copy from one to the



other by clicking on the Copy button.

You can navigate to the previous or next compiler



memory allocation setting by clicking on Prev or Next. When you're satisfied, click on OK to have the changes take effect. You may also click on Cancel if you decide not to make the changes.

Blocks and Strings

ProIcon manages execution memory automatically. It keeps track of values your program no longer needs and "collects" them when it needs space to create new values. Consequently, you can process a large amount of data as long as it doesn't all have to be kept around at one time. Some programs, however, need to keep a lot of strings and structures, such as tables of words and their counts. Such programs may need larger regions for storage than are provided by default. If this happens, you get an error message, such as

Evaluation Stack

You can increase the size of the offending region to give your program more room. Select Execution Memory from the Options menu:

Proceed as described for compiler memory: in this case, select String region from the hierarchical menu:

The current default size is shown. Deciding on a new size may take some thought and possibly experimentation. If you have a lot of RAM, you can set the value considerably larger and not worry about it.

Co-Expressions

If you don't have a lot of RAM, you may need to search for a value that works, possibly reducing the sizes of other regions. Windows also require memory. You may need to close unnecessary windows if the total amount of RAM is a problem. If you're running under MultiFinder, you may need to adjust ProIcon's partition size; see the section on MultiFinder later in this chapter. In any event, it's necessary to know something about how ProIcon uses memory.

Qualifiers

The block and string regions are where the values your program produces are kept. The other sizes refer to other aspects of program execution.

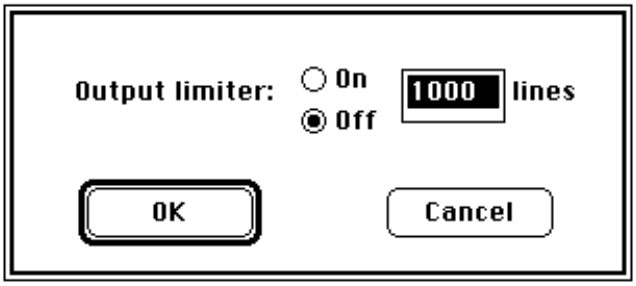
The evaluation stack is used to hold temporary values during generation and recursive procedure calls. Overflow of the evaluation stack usually indicates runaway recursion. You should look for a problem in your program if this happens. If the stack really needs to be larger, you can increase its size as for other regions. If the problem is runaway recursion, however, increasing the stack size only increases the time until overflow occurs.

Every co-expression contains its own evaluation stack. Co-expression stacks normally are smaller than the main

evaluation stack. If overflow occurs in a co-expression, it may be necessary to increase the size of co-expression blocks. (Most of the size of a co-expression block is devoted to its stack.) Since stack overflow in a co-expression may not be detected, it can cause other program malfunctions. If you're using co-expressions that perform recursive procedure calls or deeply nested generation and your program has problems, suspect such overflow.

The qualifier pointer region is used only during garbage collection to keep track of strings in your program that need to be saved. If you get an error message that indicates that this region is not large enough, increase the size. You should check your program to see if it really needs to have a lot of different strings at the same time.

There is another side to all this: If you have only a small amount of RAM, the default region sizes may be too large to even get started and you may have to reduce the default sizes to be able to run ProIcon programs at all. If this is the



case, start with a small program and reduce the string and block region sizes to smaller values until your program runs. You also can decrease the size of the evaluation stack and qualifier pointer regions if necessary. Reducing the size of co-expression blocks only has an effect if you use co-expressions. You can reduce all sizes substantially and still be able to run small programs that don't need lots of memory. However, the kinds of applications for which ProIcon is best do require lots of memory. It's impractical to try such applications without enough memory, and, while ProIcon can work over a wide range of memory, if there really is not enough memory, it will work poorly or

not at all.

Output Limiter

Because window data is held in memory, writing a large amount of data to a window may exhaust memory. ProIcon provides a way of limiting the number of text lines that are retained in a window.

The limit can be controlled from your program by selecting Window Options … from the Options menu, which gives you this dialog box:

You can turn output limiting on or off as indicated and also specify the number of lines retained when it is on.

For example, suppose you set the limit to 350 lines, and your program begins writing output to the window Untitled-3.icn. As new lines appear at the bottom of that window, the window contents scroll up and are available for review by moving the thumb in the vertical scroll bar. When the 350th line is output, ProIcon silently discards the top one-eighth of the lines in the memory file. The file now contains 306 lines of data, and the process is repeated each time the 350-line limit is reached.

Persistent Settings

Most options you set apply only during a session in the ProIcon application. Once you quit ProIcon, they are discarded. When you launch ProIcon again, they're set back to the default values.

Program options and memory settings are persistent. The paths you specify for library searches are also persistent. Persistent settings are saved in the ProIcon Profile in your system folder. You can delete this file if you want to get back to ProIcon's initial settings.

Launching Executable Files

If you hold down the option key when click-launching an executable ProIcon file, you get a dialog that allows you to set the parameter string, specify input and output files, and adjust execution memory regions.

Stopping ProIcon

You can terminate program compilation, linking, or execution by selecting Stop from the Run menu. Caution: ProIcon cannot stop all forms of program activity immediately.

If you want to suspend compilation, linking, or execution, select Pause from the Run menu. You then can resume by selecting Continue.

MultiFinder

ProIcon is MultiFinder friendly. It will run in the background if you let it. When it needs input, a small icon alternates with the symbol on the menu bar.

When running under MultiFinder, you can adjust ProIcon's memory partition by highlighting the application on the desktop and using Finder's Get Info command. The number at the bottom of the screen is the partition size.

You can launch another application from ProIcon using launch() as described in Chapter 10. If MultiFinder is active, your ProIcon program continues to run.

5

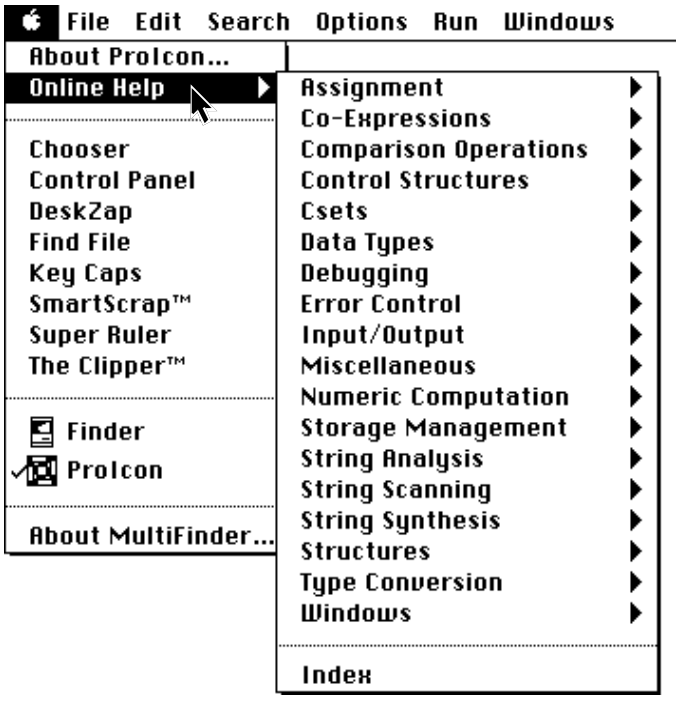
Online Help

Online Help

ProIcon provides an online help facility that allows you to get information about functions, operations, and other aspects of ProIcon without leaving the ProIcon application. Online help is not a substitute for this manual or the Icon book, but it can give you quick answers to many questions that come up while writing and debugging programs.

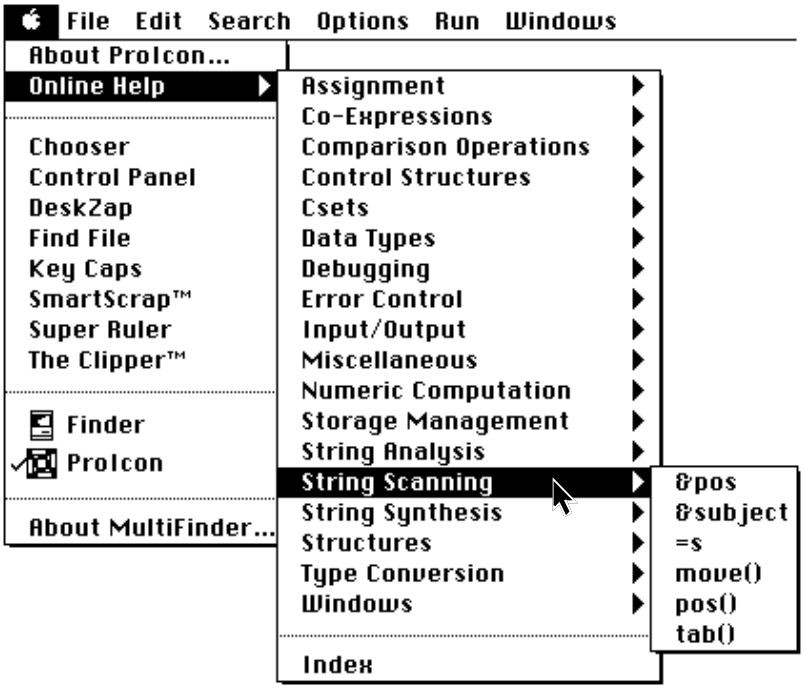
Help Menu

There are several ways that you can get online help. One is to select Online Help from the top of the menu. This gives you a two-level heirarchical menu:



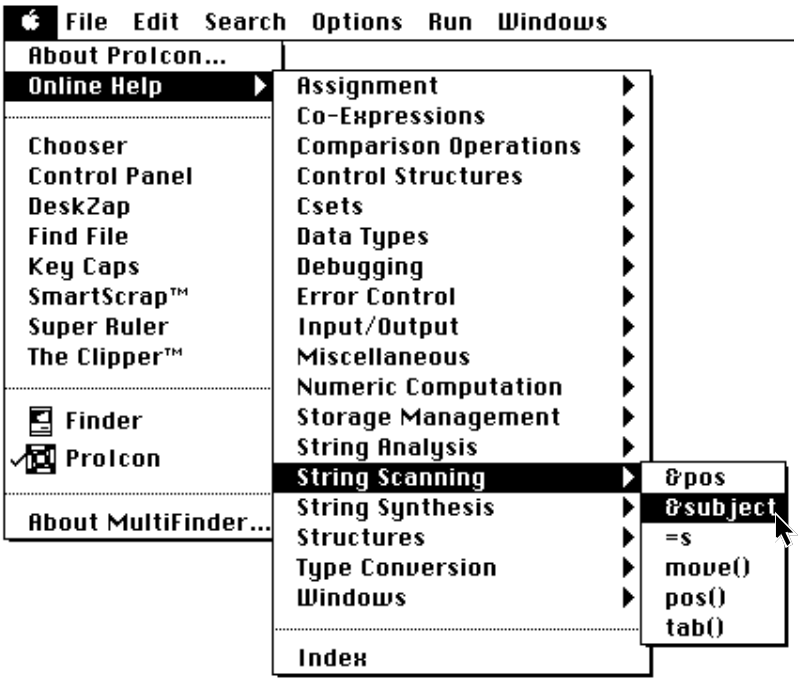
Selecting a Category

The first level, shown on the previous page, lists the categories for which help is available. The next level lists specific items. Suppose, for example, that you select String Scanning:



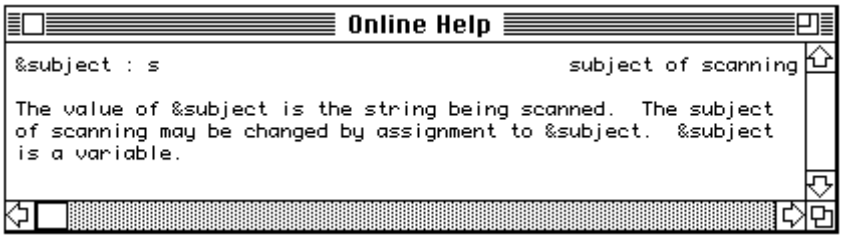
Selecting an Item

Now you can select a specific item, such as &subject:



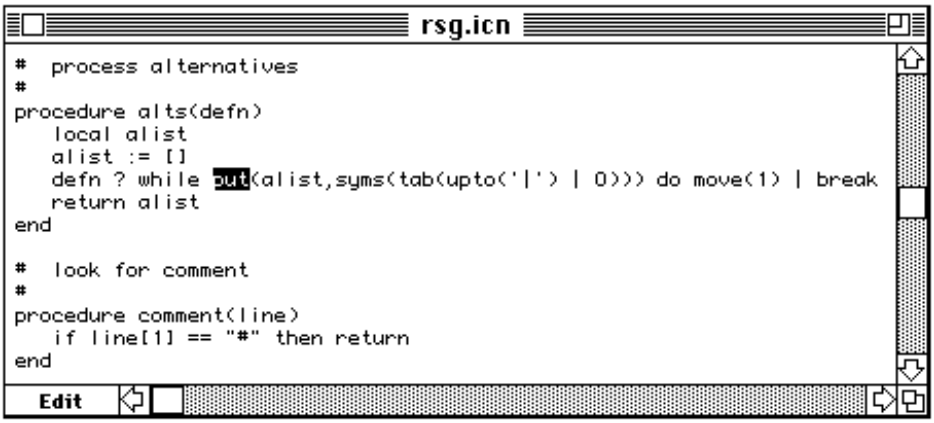
Help Windows

A window with information about &subject pops up:



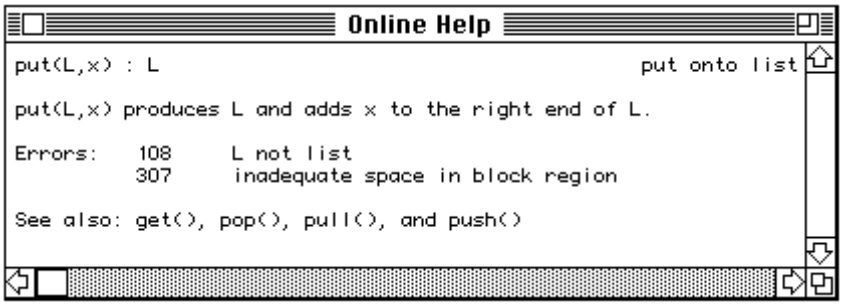
Help Lookup

Another way to get help is to select text in the currently active window, such as the name of a function in a program on which you're working. Suppose, for example, you're working on the program shown below and you want to know more about the function put. Select put by dragging over it or double clicking on it:



Now select Help Lookup from the Search menu:

A help window for put pops up:



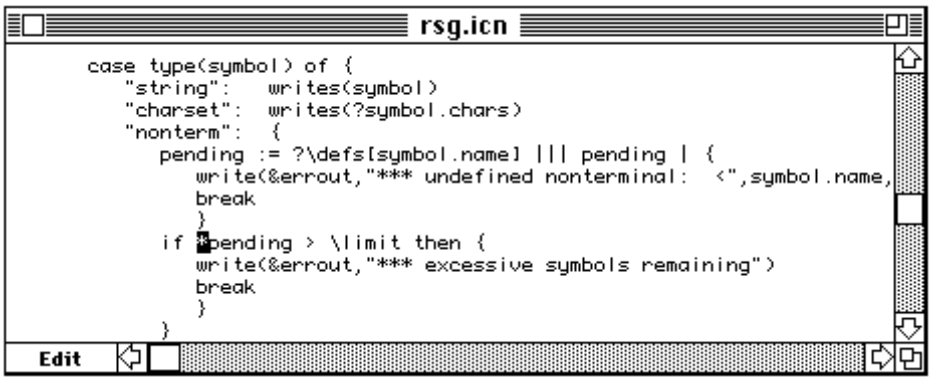
Keyboard Shortcuts

You can get online help in this way by selecting text from any window, including a help window. This is useful, for example, for following cross references such as the one for put in the window on the previous page.

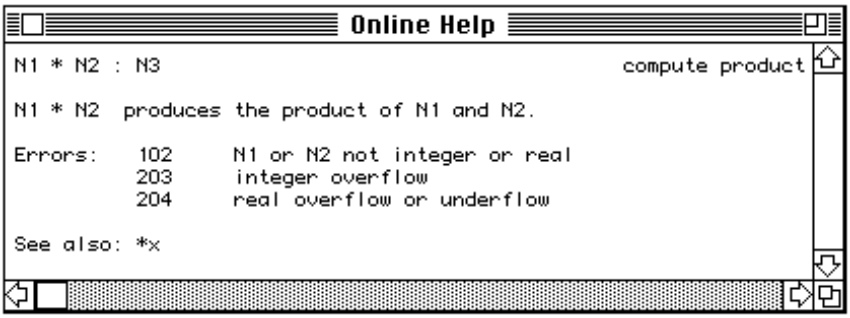
As indicated in the Search menu, ⌘-H is a keyboard shortcut for getting help. The help key on the extended keyboard does the same thing.

Help for Operators

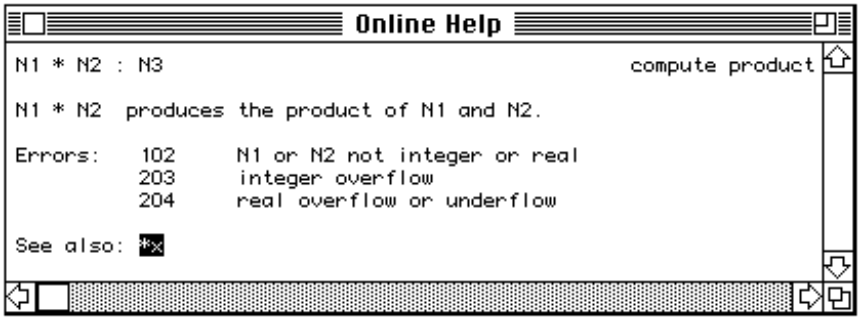
To get help for an Icon operator, select the symbol or symbols for the operator. Since some prefix and infix operators use the same symbols, the selection may be ambiguous. If it is, you get a help window for the infix operator. For example, if you select the operator *, as in



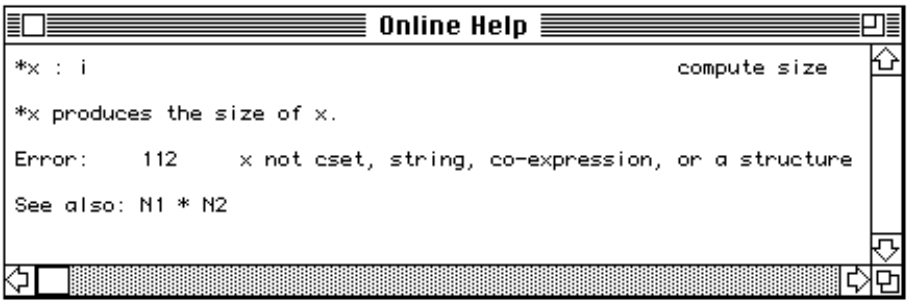
You get the following help window:



This isn't what you wanted, but it has a cross reference to the prefix operator, so you can get help for the prefix operator from there. In this case, select the operator and its operand as well; this is how the online help system distinguishes between the two cases.



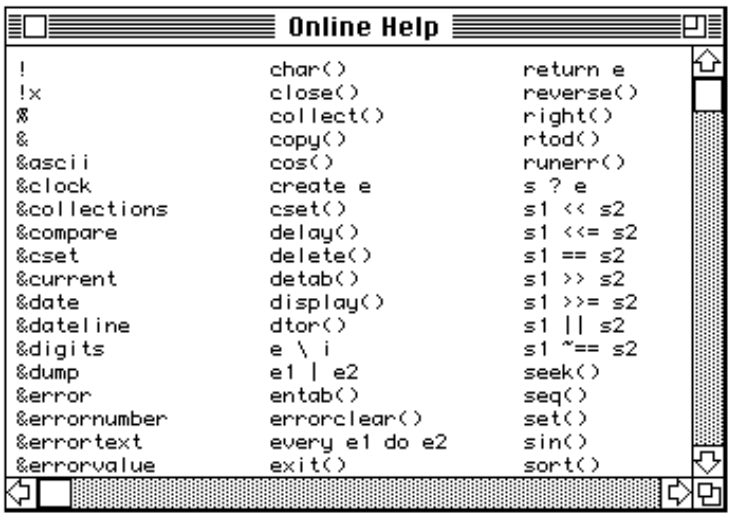
The help window for this selection is what you wanted originally:



As you see, help windows for prefix operators also contain cross references to infix operators, so you can navigate as you wish.

Help Index

You may have noticed that the last item on the first level of the help menus is Index. If you select it, you get a window that lists all the help entries:



This window is a handy way to find something if you're not quite sure what it is or if you don't know its category. You can, of course, select any help entry from the index.

6

Entering and Editing Text

Entering and Editing Text

This chapter explains text editing and window management. You'll read about:

- Editing text in windows.
- Additional editing shortcuts available when your program is reading input from the keyboard.
- Managing windows on the desktop.

ProIcon follows the Apple human interface guidelines, with a few extensions. If you're familiar with other text editors on the Macintosh, you'll feel right at home with the ProIcon editor.

The Editor

The ProIcon editor can be used to create or edit any kind of text file on a Macintosh, not just program files. There is only one restriction: The file must fit entirely in RAM.

Creating New Files

Untitled Windows

To get to the editor, launch ProIcon. A window appears with the name Untitled-1.icn as shown in Chapter 3.

If the default settings have been changed so that an untitled window does not appear, then you can select New from the File menu, or click on the New button if you were presented with a file dialog. Either gives you an untitled window. ⌘-N is a shortcut to open a new, untitled window.

ProIcon can handle up to 12 windows at once. Window 0 is always assigned to the terminal. Untitled windows start at 1, and can go up to 11.

As soon as an untitled window is open, it's ready for you to start typing into it.

Opening Existing Files

To open an existing file, select Open... from the File menu. A list of files appears, and you can select one to open through a standard file dialog. ⌘-O is the shortcut here.

File Signatures

Normally the dialog is filtered to show only files with a TEXT signature. Holding down the option key displays all files in a given folder.

Below the scrolling window with the list of files, you have a choice of All text files and .icn files only. The latter choice filters file names so that only those whose names end with the letters .icn are displayed.

If you can't find a file, you may have to click on All text files. The listed files are those in the currently selected folder, and they're in alphabetical order.

To open an existing file, double-click on its name or press the Open button when the file name is highlighted. If the file is already open in ProIcon, double-clicking on its name brings its window to the front.

Scrolling through a long list of files can be tedious, but there's a shortcut: If you type the first letter of a file's name, it brings you to that alphabetic section of the file list.

Typing Text

The cursor or insertion point is indicated by a blinking vertical line. There is also the mouse pointer, a flared vertical line (known as an I-beam) when it's in the text and an arrow when it's outside the text window.

Typed text is inserted at the current position of the cursor. If more than one character of text is selected, the typed text replaces the current selection.

The delete key deletes one character to the left of the

insertion point (it backspaces). If you have an extended keyboard, there is a forward delete key (del marked with an X inside an outlined right arrow). It deletes characters to the right of the insertion point. Holding either key down until it auto-repeats removes multiple characters to the left or right.

This is a programmer's editor, not a word-processing editor, so there is no word-wrap. If you type past the right edge of the window, the window scrolls to the right. To go back and forth, use the scroll bar at the bottom.

Selecting Text

Text may be selected by placing the mouse at one end of the desired selection, and holding the mouse button down while dragging in either direction. The selection "sticks" when the mouse button is released. Selected text is displayed by a reverse highlight — white letters on a black background. If the mouse button is pressed and released without dragging, a cursor appears between characters of text.

There are other methods of selecting text. Positioning the mouse pointer over a word and double-clicking selects the entire word; triple clicking selects the entire line.

Finally, typing ⌘-A, or choosing Select All from the Edit menu selects all text in a window.

An existing selection can be extended or shortened by holding down the shift key prior to pressing the mouse button.

Clipboard

ProIcon supports the Macintosh clipboard as a place to temporarily store text. The clipboard can hold up to 32,767 characters of text. The clipboard may be used to move text between windows, applications, desk accessories, or dialogs that accept text input.

Cut, Clear, Copy, and Paste

You can delete and place selected text on the system clipboard by typing ⌘-X or by choosing Cut from the Edit menu.

Selected text can be deleted without altering the

clipboard by pressing the delete key or by choosing Clear from the Edit menu. ProIcon also responds to the clear key on the extended keyboard.

Selected text can be copied to the clipboard without removing it from its window by using ⌘-C, or choosing Copy from the Edit menu.

Typing ⌘-V or choosing Paste from the Edit menu inserts the contents of the clipboard at the current position of the cursor. If there is highlighted text, rather than a simple cursor, the paste operation replaces the selected text with the contents of the clipboard.

Undoing Changes

If you make a change that you regret, ⌘-Z undoes the change. You can also select Undo in the Edit menu. Keep in mind that Undo has a short memory; it only remembers the last thing you did.

You can also use Undo to redo what you just undid. In the Edit menu, this is reflected in the wording. For instance, if you just cut a group of several lines, the menu choice is Undo Cut. If you perform the Undo, the previously cut lines are restored, as well as the old contents of the clipboard. The Edit menu choice changes to Redo Cut, and choosing that re-performs the original cut.

If you've got more to undo than Undo can handle — a really botched file where you want to begin again from where you started this time — use the Revert command in the File menu. Selecting Revert in the dialog that appears takes you back to the last version you saved.

On the extended keyboard, function keys F1 - F4 perform undo, cut, copy, and paste, respectively.

Closing and Saving

It's prudent to save your work every 15 minutes or so, since power failures and system crashes usually occur at the worst possible times.

To save a file without closing it, use ⌘-S or select Save from the File menu. If you haven't named your file

yet, you get a standard file dialog to prompt you for a name.

To save a file and close its window, you can click the close box at the left edge of the window's title bar. If the file has been altered since it was last saved, a dialog asks if you want the new version saved. Again, if it's untitled, go through the standard file dialog to name it.

Another way to close a file is to use ⌘-W or the Close command in the Edit menu. Both are equivalent to clicking the Close box on the window.

Saving a file does not alter its file type or creator signatures. ProIcon records information in the file's resource fork to remember font, font size, tab, and compiler options. Other resources in the file remain unchanged.

File Signatures

You can also save a file under a different name by selecting Save as... in the File menu. This launches the standard file dialog for the new name. The original file remains unchanged. The new file has a TEXT file type and ProIcon's creator signature.

Here is a good technique when you're modifying a program, and you want to keep the original around: Open Original.icn, and immediately save it as Mycopy.icn. That becomes the currently open window, and you can work there, secure in the knowledge that Original.icn is still present in unmodified form.

To save all open windows, use the Save All command in the Windows menu. This works the same way as using the Save command on each window. The Close All command closes all open windows. If any window has been modified, you are asked about saving its changes.

Moving Around

The insertion point shows where anything you type is entered. You can view other parts of your file without changing the insertion point by using the mouse on the window's scroll bars.

Drag the scroll bar "thumb" to display the portion of the file you want to view.

Click on the arrow box at the end of the scroll bar to

move the window display in the indicated direction one line at a time (vertical scroll bar), or by 10 screen pixels (horizontal scroll bar).

Click in the gray area adjacent to the thumb to move the file up or down one screen at a time (vertical scroll bar), or left and right one-half screen at a time (horizontal scroll bar).

Hold the mouse down in the either area to produce continuous scrolling after a slight delay.

If you have an extended keyboard, the home and end keys move the display to the beginning or end of the file. The page up and page down keys scroll the file by the height of the window. None of these actions alters the insertion point.

To quickly reposition the file at the insertion point after scrolling and examining another part of the file, press the enter key (not the return key).

Arrow Keys

The arrow keys on the Macintosh Plus, Macintosh SE, and Macintosh II keyboards move the insertion point, one character or one line at a time. Holding down the ⌘key with the up, down, left, or right arrow key moves the insertion point to the beginning of the file, end of the file, start of a line, or end of a line respectively.

Jumps

The Search menu has several options for moving through a window:

Jump to Top

Jump to Bottom

Jump to Line #...

These have shortcuts:

⌘-'	goes to the top of the window, like ⌘-↑
⌘-;	goes to the bottom of the window, like ⌘-↓
⌘-J	asks for a line number and when you supply it, moves the insertion point and the display to that line

Fonts and Tabs

You can select a font using the Options menu. The font selected applies to the currently open window.

The font size is also selected under Options. Those that are outlined are displayed in better quality than those that aren't. The default is 9-point Monaco.

Under Options, Tabs ... you can select the number of spaces per tab stop. The default is 4.

Only one font, font size, and tab setting can be used in a window, although different windows can have different settings. The settings are remembered when the window is saved in a file.

Indenting

When checked, the Auto Indent option indents each new line typed as much as the preceding line. To prevent an auto-indent on the next line, hold down the option key when you press return. You can disable Auto Indent by unchecking it in the Options menu.

To change the indentation for a range of lines, select them, then use Shift Left or Shift Right from the Edit menu. The keyboard shortcuts are ⌘-[and ⌘-]. The selected text is moved one tab stop left or right. If you hold down the shift key while using this command, text is moved in one-space increments instead.

Balancing

The Balance command in the Edit menu (⌘-B) extends the current selection in both directions until it encloses the smallest span of text that is balanced by parentheses ( ), brackets [ ], or braces { }.

Repeat this command to select larger portions of text. If the text is out of balance, the system beeps.

Keep in mind that the Balance command just looks for matching characters. It won't know if what it finds is relevant program code or material inside a string literal or comment.

Printing

The ProIcon editor uses the standard Macintosh print routines, and thus works with an ImageWriter or a LaserWriter.

Print ... (shortcut ⌘-P) and Page Setup ... are in the File menu.

Searching and Replacing

If you have intricate searching and replacing to do, write a ProIcon program. ProIcon is far more versatile and powerful than any text editor. For garden-variety searches and replacements, though, the ProIcon editor offers a host of convenient features.

Search-and-replace operations are found under the Search menu.

Finding

The shortcut for finding text is ⌘-F. You are asked what to look for. Type that in, and click the Find button or press return. The search proceeds forward from the current insertion point. The next instance of what you're looking for is highlighted. If it can't be found, the system beeps.

You have several options that can be checked:

Match Words	Only entire words match your search text. If Match Words is off, then searching for the also matches there and bother. If on, only the matches.
Wrap Around	Searches normally go from the current insertion point to the end of the file and stop. If Wrap Around is on, the search goes back to the beginning after it reaches the end, until it gets to where you started.
Ignore Case	Means that, for example, the and tHe both match. If Ignore case is off, they don't.

To repeat a search, use ⌘-G or Find Again in the Search menu.

Replacing

If you want to replace some, but not all, occurrences of the search string, enter a replacement string in the Replace with: part of the Find dialog box.

When the editor finds the search string, you can skip it and go to the next one with ⌘-G. If you want to replace, ⌘-R does the job. ⌘-Y performs Replace and Find Again, which can speed your work.

There is also Replace All ... in the Search menu, which does exactly that. If the Replace with: field is empty, you delete every occurrence of what's in the Find: field.

Warning: Although you can undo a single search-and-replace, you can't undo a Replace All.

As a shortcut, you can enter text that is selected in a window into the Find dialog by selecting Enter Selection in the Search menu (keyboard equivalent ⌘-E). With this method, you can initiate a search without ever bringing up the Find dialog. Just select the text you want to search for, enter ⌘-E, and enter ⌘-G to find it.

Don't Find

The Search dialog lets you proceed (the Find button) and pretend you never started (Cancel). The other button is Don't Find. It keeps the current settings, but does not initiate a search.

A good time to use this is when you realize that the insertion point isn't where you want it, but you've already entered the search and replace information. Just Don't Find, move the insertion point, then Find Again.

Nonprinting Characters

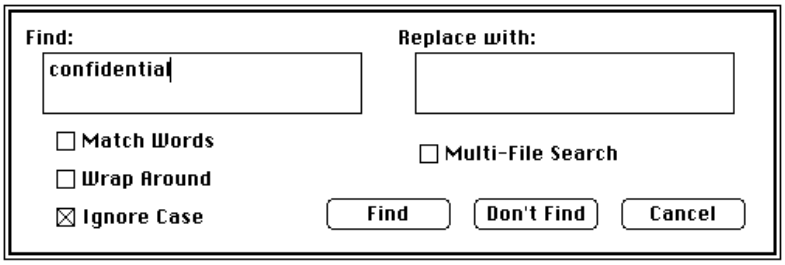
Since ProIcon uses printable ASCII characters for its source files, you won't need nonprinting characters often. If you do, you can enter the Tab and Return characters by holding down the ⌘key as you type them into the Find: and Replace with: fields.

Other nonprinting characters can be entered. Consult the Keyboard Chart in Appendix A. Any character code from 0 through 255 can be entered in this manner.

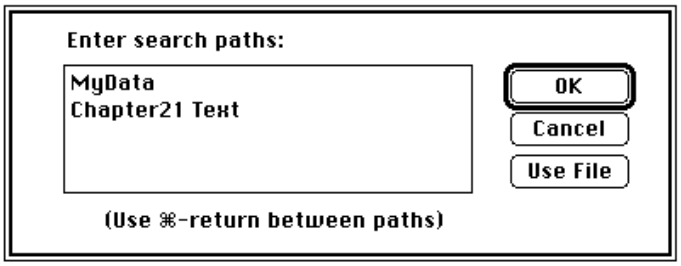
Multiple Files

The Multi-File Search option lets you search for (and optionally replace) text in more than one file. *Search paths* let you specify the names of the files to be searched.

To do this, enter your search text (and any replacement text) in the Find dialog, just as you would for a normal search of text in a window. Then check the Multi-File Search box. Suppose you want to search for the word confidential in several files. Enter the search text:



Check the Multi-File Search box. When you do, another dialog appears. Now, supply the names of the files to be examined. Suppose you want to search the files MyData and Chapter21 Text for confidential. Enter both file names, using ⌘-return between them. File names are insensitive to upper- and lowercase spellings. After entering the file names, select OK, like this:



Return to the original search dialog. When you select the Find button, the ProIcon editor searches each of the files for the desired text. If a file containing the text is found, the file is displayed and the matching text highlighted.

Find in Next File

After a successful search, the normal search continuation choices are available: Find Again, Replace, Replace and Find Again, and Replace All ... . All of these are restricted to the file just displayed.

You can continue the search into other files on your list by entering ⌘-T or by choosing Find in Next File in the Search menu.

exceed that limit. If twelve windows are open when a multi-file search is initiated or continued, ProIcon asks you to close at least one window before proceeding with the search.

Wildcards

Rather than specifying individual files, your multi-file list may include classes of file names. To do this, use wildcard characters in the names as described in Chapter 4.

Path Files

If you want avoid retyping the search paths every time you start up ProIcon, you can record them in a text file and load the search dialog with Use File as described in Chapter 4.

The Interactive Window

The Interactive window is used when your program requires interactive input from the keyboard. As characters are entered, they are echoed to this window.

All of the normal Macintosh window editing operations such as text selection, cut, copy, and paste are available to you when input is requested. You can highlight and copy text in one window, and then paste it into the input line at the end of the Interactive window.

Terminal Input Shortcuts

As a convenience, ProIcon offers some additional editing capabilities during (and only during) program input:

1. If the insertion point is anywhere on the last line, pressing return causes ProIcon to accept that line of input rather than insert a return character. The same action applies when there is a highlighted selection. Thus, after pasting in text, you can immediately press return without first having to position the insertion point at the end of line. Use ⌘-return to enter a literal return character in the input line.

2. If the insertion point lies in any of the text preceding the input line being composed, pressing return or enter moves the insertion point to the end of the input line and repositions the window accordingly.

3. If text is highlighted in any window, pressing return appends that text to the end of the input line being composed. It's as if you had selected Copy, then positioned the insertion point at the end of the input line and done a Paste.

Pressing enter instead of return also copies the text, but in addition it accepts the line for input, as if a second return had been entered after the text was copied.

Window Management

ProIcon supports all of the window regions of the standard Macintosh interface.

Window Regions

The box in the left corner of the title bar closes the window. If the text contained in the terminal window has been modified since it was last saved, ProIcon prompts you to save or discard the new contents or to cancel the Close operation. Holding down the option key while clicking in the Close box closes all windows.

Closing the Interactive window does not discard it, but merely hides it. You can make it visible again from the Window menu (described below).

The box in the right corner of the title bar is the zoom box. Clicking in this box enlarges the window to full screen size or returns it to its former size.

The scroll bars and window-resizing region in the lower right corner are entirely conventional. They let you position the file within the window and change the size of window.

If the file name ends with the suffix .icn, the window is considered runnable and contains a small status display in the lower left corner. It informs you that your program is available for editing, is being compiled, is running, has paused, or is awaiting input from the keyboard.

Window Selection

Clicking anywhere in a window or its title bar places the window in front of all other windows on the desktop.

Clicking and dragging in the title bar allows you to position the window on the desktop. Holding down the ⌘ key when clicking in the title bar allows you to move the window without bringing it to the front.

Window Arrangement

When several windows are simultaneously open, windows may become hidden behind other windows. The Window menu displays a list of names of all windows, and associates a numbered ⌘ key with the first 10. Selecting the window's name from the Window menu, or pressing the associated ⌘ key brings that window to the front. The Interactive window is permanently assigned to ⌘-0.

Selecting Stack Windows from the Window menu quickly arranges all open windows in an orderly cascade on the screen.

Tile Windows reduces the size of all windows so that they all fit on the screen, without overlap.

Holding down the option key when clicking in a window's title bar places it behind all other windows.

The Full Titles menu item attempts to display the full path name in each window's title bar and in the Windows menu. Long path names are truncated on the right.

The Zoom menu item duplicates the function of the zoom box in the title bar. It is available with a keyboard shortcut as ⌘-=.

The size, position, and current text selection are remembered with each window. This is automatic for the Interactive window but only occurs with other windows when they are saved.

Startup Options

The Startup ... option in the Windows menu allows you to determine ProIcon's launch behavior. It can start up by presenting you with a file dialog, which allows you to select the file to load; it can open a new, untitled window; or it can remain neutral, doing neither.

7

Menu Reference

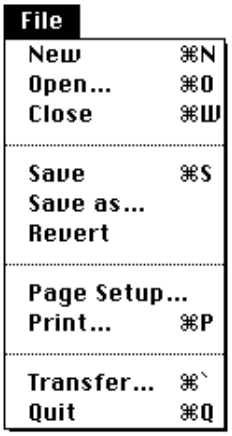
Menu Reference

This chapter summarizes ProIcon's menus. The command shortcuts shown on the menus can save you a lot of time if you use ProIcon frequently. They are summarized at the end of this chapter.

The top of the menu has two selections for ProIcon.

Select About ProIcon ... if you want to know more about the application.

Online Help opens the door to information about ProIcon that you may need when you're using ProIcon. Online help is discussed in Chapter 5.



The File menu is similar to the file menus of most Macintosh applications.

Select New to open a new window for entering a program or other text.

Select Open ... to get a dialog box that allows you to open an existing text file.

Close closes the currently active window. If it has not been saved or if it has been changed since it was last saved, you'll be asked if you want to save it.

Save saves the currently active window under its current name, while Save as ... lets you save it under a different name.

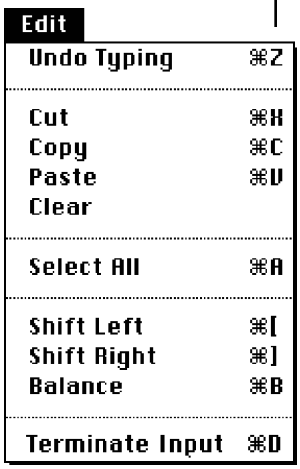
Revert goes back to the last saved version of a file and updates the currently active window accordingly.

Page Setup ... displays a dialog box that lets you determine how the contents of windows are printed.

Print ... displays a dialog box that allows you to print the contents of the currently active window.

Transfer ... allows you to transfer to another application directly from ProIcon. It displays a dialog box that lets you select the application you want.

Finally, Quit terminates the ProIcon application. If you have any windows that have been modified but not saved, you are prompted to save or discard the contents.



The Edit menu lets you do the usual Macintosh operations with the clipboard and also provides some extra facilities for use with the ProIcon editor. See Chapter 6 for detailed information.

The Undo selection lets you undo the last cut/paste or editing operation you did. What appears in that selection depends on what you did last. In the selection shown here, the last operation was the typing of text.

Cut copies the current selection to the clipboard and deletes it.

Copy does the same, but it does not delete the selection from the currently active window.

Paste copies the contents of the clipboard into the currently active window at its selection point.

Clear deletes the current selection but does not change the contents of the clipboard.

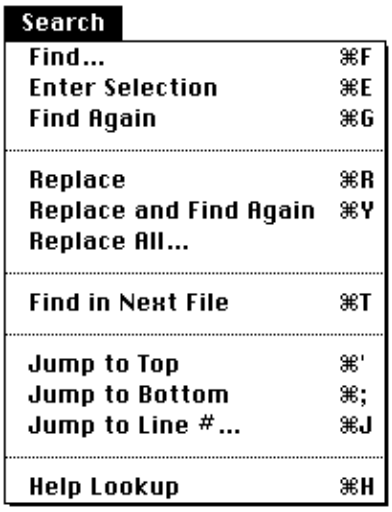
Select All selects the entire contents of the currently active window.

Shift Left and Shift Right move the selected text left or right one tab stop. If you hold down the shift key when selecting one of these, the shift is one space instead of

one tab stop.

Balance extends the current selection in both directions until it is balanced with respect to parentheses, brackets, or braces.

Terminate Input terminates keyboard input to a running program. It causes read() or reads() to fail.



The Search menu lets you locate and change window text.

Find ... displays a dialog box that allows you to specify the text you want to find. See Chapter 6 for details.

Enter Selection places the selected text in the currently active window as the text to be found.

Find Again searches for the next occurrence of the text specified for Find.

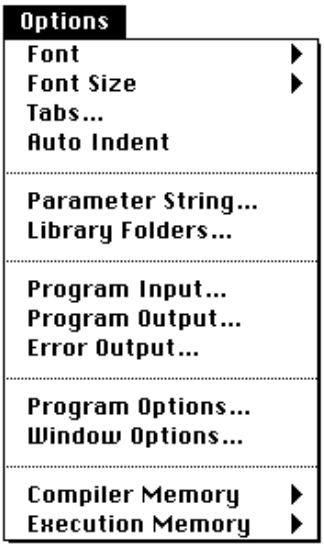
Replace allows you to replace text that is found and Replace and Find Again follows the replacement with another search.

Replace All ... replaces all occurrences of text that is found. *Warning:* you cannot undo this.

Find in Next File goes to the next file in a multi-file search. See Chapter 6 for more information.

Jump to Top and Jump to Bottom set the insertion point and window display to the top and bottom of the currently active window, respectively.

Jump to Line # ... prompts you for a line number and then



moves the insertion point and window display to that line.

Help Lookup produces a help window for the currently selected text.

The Options menu lets you change aspects of the currently active window and also lets you specify options for compiling, linking, and running Icon programs.

Select Font or Font Size to get a hierarchical menu from which you can specify the font or font size for the currently active window.

Select Tabs ... to specify the number of spaces per tab stop.

If Auto Indent is checked, each new line typed is indented as much as the preceding line.

Parameter String ... lets you specify arguments for the main procedure of your Icon program. See Chapter 5 for details.

Library Folders ... allows you to specify which folders the linker searches for intermediate files that are specified in link declarations. See Chapter 5 for details.

If you want to specify where your Icon program gets standard input, select Program Input ... . You get a dialog box that allows you to chose between the keyboard, the currently active window, or a file, in which case you get a file dialog box that allows you to chose a file.

Program Output ... is similar to Program Input ..., except it lets you chose where program output goes: to the Interactive window, to the currently active window, or to a file you chose.

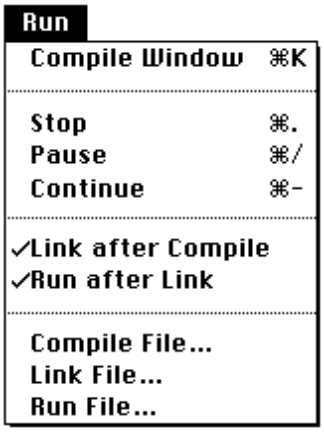
Error Output ... lets you chose where error messages go.

Program Options ... brings up a dialog box that lets you chose several options related to running Icon programs. See Chapter 5 for details.

Window Options ... lets you limit output to the currently active window and specify how windows are opened from

the desktop.

Compiler Memory gives you a hierarchical menu from



The Run menu allows you to compile and run ProIcon programs.

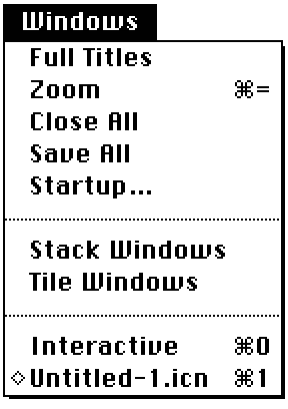
Compile Window compiles the program in the currently active window.

Stop terminates the currently executing program, while Pause causes it to suspend execution temporarily. You can continue the execution of a suspended program by selecting Continue.

If Link after Compile is checked, ProIcon goes on to link your program after compiling it. Uncheck this item if you want to stop after compilation.

Run after Link goes one step further and runs your program after it has been compiled and linked.

The last three items on the Run menu allow you to compile, link, and run existing files. If you select one of these items, you get an open dialog box from which you can chose a file.



The Windows menu lets you control several aspects of window handling.

If you check Full Titles, the full path name (or as much as fits) for a window connected to a file is displayed in the window's title bar.

Zoom duplicates the function of the zoom box in the title bar of a window.

Close All closes all windows.

Save All saves the contents of all windows that are connected to files.

Startup ... lets you determine ProIcon's launch behavior.

Stack Windows arranges all open windows in a cascade on the screen.

Tile WIndows reduces the sizes of all windows so that they do not overlap on the screen.

The bottom part of the Windows menu shows all the open windows. Select a window to bring it to the front on the screen. Windows whose contents have changed since they were saved are marked at the left with a lozenge.

Keyboard Shortcuts

Command-Key Shortcuts

Many menu items have command-key shortcuts, as shown on the menus. Here's a summary of these and other shortcuts for quick reference:

⌘-A	Select all text in window
⌘-B	Balance parentheses, brackets, or braces
⌘-C	Copy selection to clipboard
⌘-D	Terminate input (keyboard end-of-file)
⌘-E	Enter selection as search text in Find dialog
⌘-F	Find text
⌘-G	Find again
⌘-H	Help lookup
⌘-J	Jump to line
⌘-K	Compile program in window
⌘-N	Open new, untitled window
⌘-O	Open file into window
⌘-Q	Quit ProIcon
⌘-R	Replace selection
⌘-S	Save window in file
⌘-T	Find in next file
⌘-V	Paste from clipboard

⌘-W	Close window
⌘-X	Cut selection to clipboard
⌘-Y	Replace selection and find again
⌘-Z	Undo last editing operation
⌘-`	Transfer to another application
⌘-[	Shift selected lines left one tab stop
⌘-]	Shift selected lines right one tab stop
⌘-{	Shift selected lines lift one space
⌘-}	Shift selected lines right one space
⌘-'	Jump to first line in window
⌘-;	Jump to last line in window
⌘-=	Zoom window in or out
⌘-.	Stop program execution
⌘-/	Suspend program execution
⌘- –	Continue program execution
⌘-0	Select window 0
...	...
⌘-9	Select window 9
⌘-return	enter literal return in dialog
⌘-tab	enter literal tab in dialog

Function Keys

In addition, on the extended keyboard, function keys F1 - F4 perform undo, cut, copy, and paste, respectively. On all keyboards that have it, the escape key behaves like ⌘-. .

Option Key

Holding down the option key while performing certain actions produces alternatives that may be useful for one-time operations:

Press option when choosing Open in the File menu to suppress TEXT file filtering. All files types are displayed.

Press option-return when entering text in a window with auto indentation enabled to suppress auto indentation on the next line.

Press option when you start to compile or run a program to produce a pop-up dialog that allows you to make last-minute changes to the parameter string, memory allocation, and input / output specifications.

Press option when launching ProIcon from the desktop to reverse the normal startup action. If you normally get a new, untitled window, you get a file dialog instead. Conversely, if you normally get a file dialog, you get a new window.

Window Shortcuts

Manipulating Windows

Press option while clicking in the active window's go-away box to close all windows.

Press option while clicking in its title bar to place a window behind all other windows.

Press ⌘ while clicking in the window's title bar to drag a window without bringing it to the front.

8

Icon Language Overview

ICON LANGUAGE OVERVIEW

The overview of the Icon language that follows is adapted from a technical report published by the Icon Project at The University of Arizona. If you're new to Icon, this overview will get you started. If you've used Icon before, you can also use this overview to help you brush up.

Introduction

Icon is a high-level programming language with extensive facilities for processing strings and lists. Icon has several novel features, including expressions that may produce sequences of results, goal-directed evaluation that automatically searches for a successful result, and string scanning that allows operations on strings to be formulated at a high conceptual level.

Icon emphasizes high-level string processing and a design philosophy that allows ease of programming and short, concise programs. Storage allocation and garbage collection are automatic, and there are few restrictions on the sizes of objects. Strings, lists, and other structures are created during program execution and their sizes do not need to be known when a program is written. Values are converted to expected types automatically; for example, numeral strings read in as input can be used in numerical computations without explicit conversion.

Examples of the kinds of problems for which Icon is well suited are:

- text analysis, editing, and formatting
- document preparation
- symbolic mathematics
- text generation
- parsing and translation

- data laundry
- graph manipulation
- rapid prototyping

A brief description of some of the representative features of Icon is given in the following sections. This description is not rigorous and does not include many features of Icon. See the Icon book for a complete description and Chapters 9-11 of this manual for recent changes and additions to the language.

Strings

Strings of characters may be arbitrarily long, limited only by the architecture of the computer on which Icon is implemented. A string may be specified literally by enclosing it in double quotation marks, as in

```
greeting := "Hello world"
```

which assigns an 11-character string to greeting, and

```
address := ""
```

which assigns the zero-length *empty* string to address. The number of characters in a string s, its size, is given by *s. For example, *greeting is 11 and *address is 0.

Icon uses all 256 characters of the extended ASCII character set. There are escape conventions, similar to those of C, for representing characters that cannot be keyboarded.

Strings also can be read in and written out, as in

```
line := read()
```

and

```
write(line)
```

Strings can be constructed by concatenation, as in

```
element := "(" || read() || ")"
```

If the concatenation of a number of strings is to be written out, the write function can be used with several arguments to avoid actual concatenation:

```
write("(",read(),")")
```

Substrings can be formed by subscripting strings with range specifications that indicate, by position, the desired range of characters. For example,

```
middle := line[10:20]
```

assigns to middle the string of characters of line between positions 10 and 20. Similarly,

```
write(line[2])
```

writes the second character of line. The value 0 refers to the position after the last character of a string. Thus,

```
write(line[2:0])
```

writes the substring of line from the second character to the end, thus omitting the first character.

An assignment can be made to the substring of string-valued variable to change its value. For example,

```
line[2] := "..."
```

replaces the second character of line by three dots. Note that the size of line changes automatically.

There are many functions for analyzing strings. An example is

```
find(s1,s2)
```

which produces the position in s2 at which s1 occurs as a substring. For example, if the value of greeting is as given earlier,

```
find("or",greeting)
```

produces the value 8.

Character Sets

While strings are sequences of characters, *csets* are sets of characters in which membership rather than order is significant. Csets are represented literally using single enclosing quotation marks, as in

```
vowels := 'aeiouAEIOU'
```

Two useful built-in csets are &lcase and &ucase, which consist of the lowercase and uppercase letters, respectively. Set operations are provided for csets. For example,

```
letters := &lcase ++ &ucase
```

forms the cset union of the lowercase and uppercase letters and assigns the resulting cset to letters, while

```
consonants := letters  --  'aeiouAEIOU'
```

forms the cset difference of the letters and the vowels and assigns the resulting cset to consonants.

Csets are useful in situations in which any one of a number of characters is significant. An example is the string-analysis function

```
upto(c,s)
```

which produces the position in s at which any character in c occurs. For example,

```
upto(vowels,greeting)
```

produces 2. Another string-analysis function that uses csets is

```
many(c,s)
```

which produces the position in s after an initial substring consisting only of characters that occur in s. An example of the use of many is in locating words. Suppose, for example, that a word is defined to consist of a string of letters. The expression

```
write(line[1:many(letters,line)])
```

writes a word at the beginning of line. Note the use of the position returned by a string-analysis function to specify the end of a substring.

Expression Evaluation

Conditional Expressions

In Icon there are *conditional expressions* that may *succeed* and produce a result, or may *fail* and not produce any result. An example is the comparison operation

```
i > j
```

which succeeds (and produces the value of j) provided that the value of i is greater than the value of j, but fails otherwise. Similarly,

```
i > j > k
```

succeeds if j is between i and k.

The success or failure of conditional operations is used instead of Boolean values to drive control structures in Icon. An example is

```
if i > j then k := i else k := j
```

which assigns the value of i to k if the value of i is greater than the value of j, but assigns the value of j to k otherwise.

The usefulness of the concepts of success and failure is illustrated by find(s1,s2), which fails if s1 does not occur as a substring of s2. Thus

```
if i := find("or",line) then write(i)
```

writes the position at which or occurs in line, if it occurs, but does not write anything if it does not occur.

Many expressions in Icon are conditional. An example is read(), which produces the next line from the input file, but fails when the end of the file is reached. The following expression is typical of programming in Icon and illustrates the integration of conditional expressions and conventional control structures:

```
while line := read() do
  write(line)
```

This expression copies the input file to the output file.

If an argument of a function fails, the function is not called, and the function call fails as well. This "inheritance" of failure allows the concise formulation of many programming tasks. Omitting the optional do clause in while-do, the previous expression can be rewritten as

```
while write(read())
```

Generators

In some situations, an expression may be capable of producing more than one result. Consider

```
sentence := "Store it in the neighboring harbor"
find("or",sentence)
```

Here "or" occurs in sentence at positions 3, 23, and 33. Most programming languages treat this situation by selecting one of the positions, such as the first, as the result of the expression. In Icon, such an expression is a *generator* and is capable of producing all three positions.

The results that a generator produces depend on context. In a situation where only one result is needed, the first is produced, as in

```
i := find("or",sentence)
```

which assigns the value 3 to i.

If the result produced by a generator does not lead to the success of an enclosing expression, however, the generator is *resumed* to produce another value. An example is

```
if (i := find("or",sentence)) > 5 then write(i)
```

The first result produced by the generator, 3, is assigned to i, but this value is not greater than 5 and the comparison operation fails. At this point, the generator is resumed and produces the second position, 23, which is greater than 5. The comparison operation then succeeds and the value 23 is written. Because of the inheritance of failure and the fact that comparison operations return the value of their right argument, this expression can be written in the following more compact form:

```
write(5 < find("or",sentence))
```

Goal-directed evaluation is inherent in the expression-evaluation mechanism of Icon and can be used in arbitrarily complicated situations. For example,

```
find("or",sentence1) = find("and",sentence2)
```

succeeds if or occurs in sentence1 at the same position as and occurs in sentence2.

A generator can be resumed repeatedly to produce all its results by using the every-do control structure. An example is

```
every i := find("or",sentence)
   do write(i)
```

which writes all the positions at which or occurs in sentence. For the example above, these are 3, 23, and 33.

Generation is inherited like failure, and this expression can be written more concisely by omitting the optional do clause:

```
every write(find("or",sentence))
```

There are several built-in generators in Icon. One of the most frequently used of these is

```
i to j
```

which generates the integers from i to j. This generator can be combined with every-do to formulate the traditional for-style control structure:

```
every k := i to j do
   square(k)
```

This expression can be written more compactly as

```
every square(i to j)
```

There are several other control structures related to generation. One is *alternation*,

expr1 | *expr2*

which generates the results of *expr1* followed by the results of *expr2*. Thus,

```
every write(find("or",sentence1) | find("or",sentence2))
```

writes the positions of "or" in sentence1 followed by the positions of "or" in sentence2. Again, this sentence can be written more compactly by using alternation in the second argument of find:

```
every write(find("or",sentence1 | sentence2))
```

Another use of alternation is illustrated by

```
(i | j | k) = (0 | 1)
```

which succeeds if any of i, j, or k has the value 0 or 1.

String Scanning

The string analysis and synthesis operations described earlier work best for relatively simple operations on strings. For complicated operations, the bookkeeping involved in keeping track of positions in strings becomes burdensome and error prone. In such cases, Icon has a string scanning facility that is analogous in many respects to pattern matching in SNOBOL4. In string scanning, positions are managed automatically and attention is focused on a current position in a string as it is examined by a sequence of operations.

The string scanning operation has the form

s ? *expr*

where s is the *subject* string to be examined and *expr* is an expression that performs the examination. A position in the subject, which starts at 1, is the focus of examination.

Matching functions change this position. The matching function move(i) moves the position by i and produces the substring of the subject between the previous and new positions. If the position cannot be moved by the specified amount (because the subject is not long enough), move(i) fails. A simple example is

```
line ? while write(move(2))
```

which writes successive two-character substrings of line, stopping when there are no more characters.

Another matching function is tab(i), which sets the position in the subject

to i and also returns the substring of the subject between the previous and new positions. For example,

```
line ? if tab(10) then write(tab(0))
```

first sets the position in the subject to 10 and then to the end of the subject, writing line[10:0]. Note that no value is written if the subject is not long enough.

String analysis functions such as find can be used in string scanning. In this context, the string that they operate on is not specified and is taken to be the subject. For example,

```
line ? while write(tab(find("or")))
   do move(2)
```

writes all the substrings of line prior to occurrences of "or ". Note that find produces a position, which is then used by tab to change the position and produce the desired substring. The move(2) skips the "or " that is found.

Another example of the use of string analysis functions in scanning is

```
line ? while tab(upto(letters)) do
   write(tab(many(letters)))
```

which writes all the words in line.

As illustrated in the examples above, any expression may occur in the scanning expression. Unlike SNOBOL4, in which the operations that are allowed in pattern matching are limited and idiosyncratic, string scanning is completely integrated with the rest of the operation repertoire of Icon.

Structures

Icon supports several kinds of structures that consist of aggregates of values with different organizations and access methods. Lists are linear structures that can be accessed both by position and by stack and queue functions. Sets are collections of arbitrary values with no implied ordering. Tables provide an associative-lookup mechanism.

Lists

Lists in Icon are sequences of values of arbitrary types. Lists are created by enclosing the lists of values in brackets. An example is

```
car1 := ["buick","skylark",1978,2450]
```

in which the list car1 has four values, two of which are strings and two of which are integers. Note that the values in a list need not all be of the same type. In fact, any kind of value can occur in a list — even another list, as in

```
inventory := [car1,car2,car3,car4]
```

Lists also can be created by list(i,x), which creates a list of i values, each of which has the value x.

The values in a list can be referenced by position much like the characters in a string. Thus

```
car1[4] := 2400
```

changes the last value in car1 to 2400. A reference that is out of the range of the list fails. For example,

```
write(car1[5])
```

fails.

The values in a list L are generated by !L. Thus

```
every write(!L)
```

writes all the values in L.

Lists can be manipulated like stacks and queues. The function push(L,x) adds the value of x to the left end of the list L, automatically increasing the size of L by one. Similarly, pop(L) removes the leftmost value from L, automatically decreasing the size of L by one, and produces the removed value.

A list value in Icon is a pointer (reference) to a structure. Assignment of a structure in Icon does not copy the structure itself but only the pointer to it. Thus,

```
demo := car1
```

causes demo and car1 to reference the same list. Graphs with loops can be constructed in this way. For example,

```
node1 := ["a"]
node2 := [node1,"b"]
push(node1,node2)
```

constructs a structure that can be pictured as follows:

Sets

Sets are collections of values. A set is obtained from a list by set(L), where L is a list that contains the members of the set. For example,

```
S := set([1,"abc",[ ]])
```

assigns to S a set that contains the integer 1, the string "abc", and an empty list. An empty set is created by set().

The operations of union, intersection, and difference can be performed on sets. The function member(S,x) succeeds if x is a member of the set S but fails otherwise. The function insert(S,x) adds x to the set S, while delete(S,x) removes x from S. A value only can occur once in a set, so insert(S,x) has no effect if x is already in S. The operator !S generates the members of S.

A simple example of the use of sets is given by the following segment of code, which lists all the different words that appear in the input file:

```
words := set()
while line := read() do
   line ? while tab(upto(letters)) do
      insert(words,tab(many(letters)))
every write(!words)
```

Tables

Tables are sets of pairs of values, a *key* and a corresponding *value*. The keys and values may be of any type. The value for any key is looked up automatically. Thus, tables provide associative access in contrast with the positional access to values in lists.

A table is created by an expression such as

```
symbols := table(x)
```

which assigns to symbols a table that has the default value x. The default value is used for new keys. Subsequently, symbols can be referenced by any key, such as

```
symbols["there"] := 1
```

which associates the value 1 with the key "there" in symbols.

Tables grow automatically as new keys are added. For example, the following program segment produces a table containing a count of the words that appear in the input file:

```
words := table(0)
```

```
while line := read() do
  line ? while tab(upto(letters)) do
    words[tab(many(letters))] +:= 1
```

Here the default value for each word is 0 and +:= is an augmented assignment operation that increments the values by one.

A list can be obtained from a table by the function sort(t,i). The form of the list depends on the value of i. For example, if i is 3, the list contains alternate keys and values of t. An example of sorting is:

```
wordlist := sort(words,3)
while write(pop(wordlist)," : ",pop(wordlist))
```

which writes the words and their counts from words.

Procedures

An Icon program consists of a sequence of procedures. An example of a procedure is

```
procedure max(i,j)
  if i > j then return i else return j
end
```

where the name of the procedure is max and its formal parameters are i and j. The return expressions return the value of i or j, whichever is larger.

Procedures are called like functions. Thus,

```
k := max(*s1,*s2)
```

assigns to k the size of the longer of the strings s1 and s2.

A procedure also may generate a sequence of values by suspending instead of returning. In this case, a result is produced as in the case of a return, but the procedure can be resumed to produce other results. An example is the following procedure that generates the words in the input file.

```
procedure genword()
  letters := &lcase ++ &ucase
  while line := read() do
    line ? while tab(upto(letters)) do {
      word := tab(many(letters))
      suspend word
      }
end
```

The braces enclose a compound expression.

Such a generator can be used in the same way that a built-in generator is used. For example

```
every word := genword() do
   if find("or",word) then write(word)
```

writes only those words that contain the substring "or".

An Example

The following program, which produces a concordance of the words from an input file, illustrates typical Icon programming techniques. Although not all of the features in this program are described in previous sections, the general idea should be clear.

```
procedure main()

   letters := &lcase ++ &ucase
   words := table()
   maxword := lineno := 0

   while line := read() do {
      lineno +:= 1
      write(right(lineno,6),"  ",line)
      line := map(line)                    # fold to lowercase
      line ? while tab(upto(letters)) do {
         word := tab(many(letters))
         if *word < 3 then next            # skip short words
         maxword <:= *word                 # keep track of longest word
         /words[word] := set()             # if it's a new word, start set
         insert(words[word],lineno)        # add the line number
         }
      }
   write()
   wordlist := sort(words,3)                # sort by words
   while word := get(wordlist) do {
      lines := ""                           # build up line numbers
      numbers := sort(get(wordlist))
      while lines ||:= get(numbers) || ", "
      write(left(word,maxword + 2),": ",lines[1:-2])
      }
end
```

The program reads a line, writes it out with an identifying line number, and then processes every word in the line. Words less than three characters long are considered to be "noise" and are discarded. The table words contains sets of line numbers for each word. The first time a word is encountered, there is no set for it (tested by /words[word]). In this case, a new set is created. The current line number is appended to the set for the word in any event.

After the input file has been read, the table of words is sorted (the corresponding values are sets of line numbers). For each word, its set is sorted and the word and line numbers where it occurs are written out. For example, if the input file is

```
        On the Future!–how it tells
        Of the rapture that impells
       To the swinging and the ringing
        Of the bells, bells, bells–
     Of the bells, bells, bells, bells,
           Bells, bells, bells–
  To the rhyming and the chiming of the bells!
```

the output is

```
1         On the Future!-how it tells
2         Of the rapture that impells
3        To the swinging and the ringing
4         Of the bells, bells, bells-
5      Of the bells, bells, bells, bells,
6              Bells, bells, bells-
7   To the rhyming and the chiming of the bells!

and       : 3, 7
bells     : 4, 5, 6, 7
chiming   : 7
future    : 1
how       : 1
impells   : 2
rapture   : 2
rhyming   : 7
ringing   : 3
swinging  : 3
tells     : 1
that      : 2
the       : 1, 2, 3, 4, 5, 7
```

It is easy to make this program more sophisticated. For example, a dictionary of words to be ignored could be added as a set. With a little more work, the output format could be made more attractive, and so on.

9

Version 8 of Icon

Version 8 of Icon

A complete description of Icon is contained in *The Icon Programming Language* (Prentice-Hall), by Ralph E. Griswold and Madge T. Griswold. The first edition of this book (1983) describes Version 5 of Icon, while the second edition (1990) describes Version 8. Version 2.0 of ProIcon corresponds to Version 8 of Icon. This chapter describes briefly the changes in Icon between Versions 5 and 8 and is provided for persons who have the first edition of the book but not the second. If you have the second edition of the book, you can skip this chapter, although you may wish to check the list of known bugs and limitations in Version 8 of Icon that are listed at the end of this chapter.

The descriptions that follow are keyed to the first edition — its chapter and page numbers.

Chapter 3 — Numbers

Page 22, Large-Integer Arithmetic: There is no limit on the magnitude of integers produced by integer arithmetic.

Page 23, Bit Operations and Mathematical Functions: There are five functions that operate on integers at the bit level:

iand(i,j)	Produces the bit-wise and of i and j.
ior(i,j)	Produces the bit-wise inclusive or of i and j.
ixor(i,j)	Produces the bit-wise exclusive or of i and j.
icom(i)	Produces the bit-wise one's complement of i.
ishift(i,j)	Produces the result of shifting i by j positions. If j is positive, the shift is left, while if j is

negative, the shift is right. Vacated bit positions are filled with zeros.

Page 23, Mathematical Functions: The following functions perform trigonometric computations:

sin(r)	sine of r
cos(r)	cosine of r
tan(r)	tangent of r
asin(r)	arc sine of r
acos(r)	arc cosine of r
atan(r)	arc tangent of r
atan(r1,r2)	arc tangent of r1 / r2

In all cases, angles are given in radians. There are two forms of arguments for atan, depending on whether or not the second argument is supplied.

The following functions convert between radians and degrees:

dtor(r)	the radian equivalent of r given in degrees
rtod(r)	the degree equivalent of r given in radians

The following functions perform mathematical calculations:

sqrt(r)	square root of r
exp(r)	*e* raised to the power r
log(r1,r2)	logarithm of r1 to the base r2 (default *e*)

Chapter 4 — Character Sets and Strings

Page 25, Csets: The keyword &letters consists of the 52 upper- and lowercase letters. The keyword &digits consists of the ten digits. It is provided as a convenience; '0123456789' works just as well.

Page 27, Strings: The function char(i) produces the one-character string corresponding to i. The function ord(s) produces the integer correspond-

ing to the one-character string string(s).

Page 37, Tabular Material: Two functions deal with tabs in textual material:

```
entab(s,i1,i2,...,in)
detab(s,i1,i2,...,in)
```

The function entab(s,i1,i2,...,in) produces a string obtained by replacing runs of consecutive spaces (blanks) in s by tab characters. There is an implicit tab stop at 1 to establish the interval between tab stops. The remaining tab stops are at i1, i2, ..., in. Additional tab stops, if necessary, are obtained by repeating the last interval. If you do not specify the tab stops, the interval is 8 with the first tab stop at 9.

For the purposes of determining positions, printing characters have a width of 1, "\b" has a width of –1, and "\r" and "\n" restart the counting of positions. Nonprinting characters (decimal codes 0-31 and 127 for ProIcon) have zero width.

A lone space is never replaced by a tab character, but a tab character may replace a single space that is part of a longer run.

The function detab(s,i1,i2,...,in) produces a string obtained by replacing each tab character in s by one or more spaces. Tab stops are specified in the same way as for entab.

Chapter 5 — Structures

Page 48, Sets: A set is an unordered collection of values. Sets have many of the properties normally associated with sets in the mathematical sense.

The function set(L) creates a set that contains the distinct elements of the list L. For example, set(["abc",3]) creates a set with two members, "abc" and 3. If the argument to set is omitted, an empty set is created.

Any specific value can occur only once in a set. For example, set([1,2,3,3,1]) creates a set with the three members 1, 2, and 3.

There are several operations on sets. The function member(S,x) succeeds and returns x if x is a member of the set S, but fails otherwise. Therefore,

```
member(S1,member(S2,x))
```

succeeds if x is a member of both S1 and S2. The function insert(S,x) inserts x into the set S and returns S. The function delete(S,x) deletes

the member x from the set S and returns S. The functions insert(S,x) and delete(S,x) always succeed, whether or not x is in S. This allows their use in loops in which failure may occur for other reasons. For example,

```
S := set()
while insert(S,read())
```

builds a set that consists of the (distinct) lines from the standard input file.

The operations

```
S1 ++ S2
S1 ** S2
S1 -- S2
```

create the union, intersection, and difference of S1 and S2, respectively. In each case, the result is a new set.

These operations apply both to sets and csets. There is no automatic type conversion between csets and sets; the result of the operation depends on the types of the arguments. For example,

```
'aeiou' ++ 'abcde'
```

produces the cset 'abcdeiou', while

```
set([1,2,3]) ++ set([2,3,4])
```

produces a set that contains 1, 2, 3, and 4.

The size of a set (the number of members in it), is given by *S. ?S produces a randomly selected member of S and !S generates the members of S. The function sort(S) produces a list containing the members of S in sorted order.

Pages 56-58, Operations on Tables: The word "key" is used in this manual for the value used to subscript a table. The first edition of *The Icon Programming Language* uses the words "entry value". The two are synonymous.

The function key(T) generates the keys in table T. (Without this function, the only way to find the keys in a table is to sort the table and pick the keys out of the resulting list.) For example,

```
every write(key(T))
```

writes all the keys in T.

Given the keys, it is possible to get the corresponding values, as in

```
every x := key(T) do
```

```
write(image(x)," : ",image(T[x]))
```

which writes the keys in T and their corresponding values.

The functions member, insert, and delete apply to tables as well as sets. The function member(T,x) succeeds if x is a key for an element in the table T, but fails otherwise. The function insert(T,x,y) inserts into table T an element with key x and value y. If there already was a key x in T, its corresponding value is changed. Note that insert has three arguments when used with tables, as compared to two when used with sets. An omitted third argument defaults to the null value.

The function delete(T,x) removes the element with key value x from T. If x is not a key in T, no operation is performed; delete succeeds in either case.

There are several options for sort(T,i). The form of the result produced and the sorting order depends on the value of i:

If i is 1 or 2, the size of the sorted list is the same as the size of the table. Each value in the list is itself a list of two values: a key and the corresponding value. If i is 1, these lists are in the sorted order of the keys. If i is 2, the lists are in the sorted order of the corresponding values. If i is omitted, 1 is assumed.

If i is 3 or 4, the size of the sorted list is twice the size of the table and the values in the list are successive keys and corresponding values for the elements in the table. If i is 3, the values are in the sorted order of the keys. If i is 4, the values are in the sorted order of the corresponding values. For example, the following program prints a count of word occurrences in the input file, using a procedure tabwords that produces a table of words and their counts:

```
procedure main()
  wlist := sort(tabwords(),3)              # get sorted list
  while write(get(wlist)," : ",get(wlist))
end
```

In this example, get obtains the key first and then its corresponding value. The list is consumed in the process, but it is not needed for anything else.

Chapter 6 — Data Types

Page 62, Type Codes: The initial letter of a type name is used to indicate its type in this manual. In cases where two types begin with the same letter, uppercase letters are used to avoid ambiguity. Uppercase letters are also used for structure types.

co-expression	C
cset	c
file	f
integer	i
list	L
null	n
procedure	p
real	r
set	S
string	s
table	T
numeric	N
record	R
any structure	X
any type	x

Page 67, Sorting: Sets sort after lists but before tables. Csets, like strings, sort in nondecreasing lexical order. Within one structure type, values are sorted in order of the time of their creation, with the oldest last.

Chapter 7 — Procedures

Page 69, Scope Declarations: The reserved word dynamic, which was synonymous with local, is no longer available.

Page 73, Procedures with a Variable Number of Arguments: A procedure can be made to accept a variable number of arguments by appending [ ] to the last (or only) parameter in the parameter list. An example is:

```
procedure sum(a,b,c[ ])
   total := a + b
   i := 0
   while total +:= c[i +:= 1]
   return total
end
```

If called as sum(1,2,3,4,5), the parameters have the following values:

a	1
b	2
c	[3,4,5]

The last parameter always contains a list. This list consists of the arguments not used by the previous parameters. If the previous parameters use up all the arguments, the list is empty. If there are not enough

arguments to satisfy the previous parameters, the null value is used for the remaining ones, but the last parameter still contains the empty list.

Page 74, Invocation with a List of Values: The operation p!L invokes p with the arguments in the list L. The example p![1, 2, 3] is equivalent to p(1, 2, 3). The operation p!L has high precedence and associates to the left.

Chapter 9 — Input and Output

Page 93, Values Returned by Output Functions: The last argument written is returned by write and writes, but it is not converted to a string.

Page 93, Random-Access Input and Output: There are two functions related to random-access input and output, in which data does not have to be read or written in sequential order.

The function seek(f,i) seeks to position i in file f. As with other positions in Icon, a nonpositive value of i can be used to reference a position relative to the end of f. The second argument defaults to 1.

Note: The Icon form of position identification is used; the position of the first character of a file is 1, not 0 as it is in some other random-access facilities.

The function where(f) returns the current byte position in the file f.

Page 93, Keyboard Functions: There are three functions for reading input from the keyboard:

getch()	gets one character from the keyboard
getche()	gets one character and echoes it to the screen
kbhit()	succeeds if there is a keyboard character to read, but fails otherwise

Page 93, File Manipulation: Files can be removed or renamed during program execution. The function remove(s) removes the file named s. Subsequent attempts to open the file fail, unless it is created anew. If the file is open, s is not removed. remove(s) fails if it is unsuccessful.

The function rename(s1,s2) causes the file named s1 to be henceforth known by the name s2. The file named s1 is effectively removed. If a file named s2 exists prior to the renaming, the renaming is not performed. rename(s1,s2) fails if unsuccessful, in which case if the file existed previously it is still known by its original name.

Chapter 10 — Miscellaneous Operations

Page 97, String Invocation: A string-valued expression that corresponds to the name of a procedure, function, or operation can be used in place of the procedure, function, or operation in an invocation expression. For example, "numeric"(x) produces the same call as numeric(x) and "–"(i,j) is equivalent to i – j.

In the case of operator symbols with unary and binary forms, the number of arguments determines the operation. Thus, "–"(i) is equivalent to –i. Although to-by is represented with reserved words, it is an operation. It can be invoked by the string name "..." . Thus, "..."(1,10,2) is equivalent to 1 to 10 by 2. Similarly, range specifications are represented by ":", so that ":"(s,i,j) is equivalent to s[i:j] The subscripting operation is available with the string name "[ ]". Thus, "[ ]"(&lcase,3) produces "c".

Defaults are not provided for omitted or null-valued arguments in string invocation. Consequently, "..."(1,10) results in a run-time error.

Arguments to operators invoked by string names are dereferenced. As a result, string invocation for assignment operations is ineffective and results in error termination.

String names are not available for control structures such as alternation. Field references, of the form

expr . fieldname

also are not operations in the ordinary sense and are not available via string invocation. In addition, conjunction is not available via string invocation, since no operation is actually performed.

String names for procedures are available through global identifiers. The names of functions, such as numeric, are global identifiers. Similarly, any procedure-valued global identifier may be used as the string name of a procedure. Thus, in

```
global q

procedure main()
   q := p
   "q"("hi")
end
procedure p(s)
   write(s)
end
```

the procedure p is invoked via the global identifier q.

The function proc(x,i) converts x to a procedure, function, or operation if possible. If x is procedure-valued, its value is returned unchanged. If the

The function args(p) produces the number of arguments expected by p. The value is –1 for a function that accepts a variable number of arguments. For a procedure declared with a variable number of arguments, the value is the negative of that number.

Page 99, String Images of Co-Expressions: The image of a co-expression includes, in parentheses, an identifying number and the number of times it has been activated. Identifying numbers start with 1 for &main and increase as new co-expressions are created. For example,

```
image(&main)
```

produces

```
co–expression_1(1)
```

assuming &main has not been activated since its initial activation to start program execution.

Page 99, Images of Structures: Structures have serial numbers that appear in their string images following the type name and an underscore. For example, image([1, 4, 9, 16]) produces a result such as "list_10(4)".

Page 100, Co-Expression Tracing: Tracing shows co-expression activation and return as well as procedure activity. display(i,f) shows an image of the current co-expression in addition to the values of variables.

Page 102, Variables and Names: The function name(v) returns the string name of the variable v. The string name of a variable or keyword is as it appears in the program. The string name of a subscripted string-valued variable consists of the name of the variable and the subscript, as in "line[2 +:3]". The string name of a list or table consists of the data type and the subscripting expression, as in "list[3]". The string name of a record field reference consists of the record type and field names, separated by a period, as in "complex.r".

The function variable(s) produces the variable for the identifier or keyword whose name is s.

Chapter 11 — Generators

Page 110, Integer Sequences: The function seq(i,j) generates an infinite sequence of integers starting at i with increments of j; i and j default to 1. For example, seq() generates 1, 2, 3, ...

Page 116, suspend-do**:** A do clause is allowed with suspend:

```
suspend expr1 do expr2
```

If the do clause is present, *expr2* is evaluated if the suspending procedure is resumed. Next, *expr1* is resumed. If it produces another result, the procedure suspends again. In this sense, suspend is very similar to every, the difference being that suspend causes the procedure in which it occurs to return a value.

Chapter 12 — String Scanning

Page 130, String Scanning: The values of &subject and &pos are restored when a scanning operation is exited by a break, next, return, fail, or suspend expression. If a scanning expression that is exited by suspend is resumed, the values of &subject and &pos are restored to the values they had before the suspension.

Chapter 13 — Co-Expressions

Page 138, Programmer-Defined Control Structures: To make programmer-defined control operations easier to use, there is an alternative syntax for procedure invocation in which the arguments are passed in a list of co-expressions. This syntax uses braces in place of parentheses:

```
p{expr1, expr2, ..., exprn}
```

is equivalent to

```
p([create expr1, create expr2, ..., create exprn])
```

Using this facility, a procedure to generate the interleaved results of several generators can be written as follows:

```
procedure Inter(L)
  suspend |@!L
end
```

For example,

```
Inter{1 to 3,6 to 10}
```

generates 1, 6, 2, 7, 3, 8, 9, and 10.

In this form, Inter can be called with an arbitrary number of arguments.

Page 138, Current Co-Expression: The value of &current is the currently executing co-expression.

Appendix A — Syntax

Reserved Words: The reserved words dynamic and external are no longer available.

The character pairs $(, $), $<, and $> are equivalent to {, }, [, and], respectively, in program text. These character pairs are useful when writing programs that are to be run on IBM 370 systems, whose input and output devices may not support braces and brackets.

Program Location Information: A comment that begins at the beginning of a line and has the form

#line *n* "*f*"

changes the current source-program line number and file name recorded by the Icon translator to *n* and *f*, respectively. This information is used, for example, in error messages.

Appendix B — Machine Dependencies and Limits

There is no longer a distinction between short and long integers.

There is no limit on the length of a string that can be read.

Appendix C — Running an Icon Program

Linking: The inclusion of intermediate files during linking can be specified in a source file by using a link declaration, which has the form

link *intermediate file names*

For example, if rsg.icn contains the declaration

```
link lib
```

then lib.u1 and lib.u2 are included when the executable file for rsg is produced. Note that the suffixes are not used in link declarations.

Several files can be specified in a comma-separated list, as in

```
link lib, listpak, timer
```

which specifies the linking of lib.u1, listpak.u1, timer.u1, and the corresponding .u2 files. File names that do not satisfy the syntax of Icon identifiers must be enclosed in quotation marks. An example is

```
link "set-up"
```

Storage Management: Storage is allocated automatically during the execution of an Icon program, and garbage collection is performed automatically to reclaim storage for subsequent reallocation. An Icon programmer normally need not worry about storage management. However, in applications that require a large amount of storage or that must operate in a limited amount of memory, some knowledge of the storage management process may be useful.

Icon has three storage regions: static, string, and block. (Some implementations, including ProIcon, do not have a static region.) The keyword &collections generates four values associated with garbage collection: the total number since program initiation, the number triggered by static allocation, the number triggered by string allocation, and the number triggered by block allocation. The keyword &regions generates the current sizes of the static, string, and block regions. The keyword &storage generates the current amount of space used in the static, string, and block regions. The value given for the static region presently is not meaningful.

Run-Time Errors: When a run-time error occurs, a diagnostic message is produced indicating the nature of the error, where in the program the error occurred, and, when possible, the offending value. Next, a trace back of procedure calls is given, followed by the offending expression.

For example, suppose the following program is contained in the file max.icn:

```
procedure main()
   i := max("a",1)
end

procedure max(i,j)
   if i > j then i else j
end
```

The execution of this program produces the following output:

```
Run-time error 102
File max.icn; Line 6
numeric expected
offending value: "a"
Trace back:
   main()
   max("a",1) from line 2 in max.icn
   {"a" > 1} from line 6 in max.icn
```

A complete list of run-time error messages is contained in Appendix C.

Error Conversion: Most run-time errors can be converted to expression failure, rather than causing termination of program execution.

If the value of &error is zero (its initial value), errors cause program termination as shown above. If the value of &error is nonzero, errors are treated as failure of expression evaluation and &error is decremented. For example, if the value of &error had been nonzero when the expression i > j was executed in the example above, the expression simply would have failed.

There are a few errors that cannot be converted to failure: floating-point overflow and underflow, stack overflow, and errors during program initialization.

When an error is converted to failure, the value of &error is decremented and the values of three other keywords are set:

- &errornumber is the number of the error (for example, 101).
- &errortext is the error message (for example, integer expected).
- &errorvalue is the offending value. Reference to &errorvalue fails if there is no specific offending value.

A reference to any of these keywords fails if there has not been an error.

The function errorclear() removes the indication of the last error. Subsequent references to the keywords above fail until another error occurs.

The keywords &file and &line contain, respectively, the name of the file and line number in that file for the currently executing expression.

Error conversion is illustrated by the following procedure, which could be used to process potential run-time errors:

```
procedure ErrorCheck()
   write("\rRun-time error ",&errornumber)
   write("File ",&file,"; Line ",&line)
   write(&errortext)
   write("offending value: ",image(&errorvalue))
   writes("\rDo you want to continue? (n)")
   if map(read()) == ("y" | "yes") then return
   else exit()
end
```

For example,

```
   &error := –1
   ...
   write(s) | ErrorCheck()
```

could be used to check for an error during writing, while

```
   (L := sort(T,3)) | ErrorCheck()
```

could be used to detect failure to sort a table into a list (for lack of adequate storage).

A run-time error can be forced by the function runerr(i,x), which causes program execution to terminate with error number i as if a corresponding run-time error had occurred. If i is the number of a standard run-time error,

the corresponding error text is printed; otherwise no error text is printed. The value of x is given as the offending value. If x is omitted, no offending value is printed.

This function makes it possible for library procedures to terminate in the same fashion as built-in operations. It is advisable to use error numbers for programmer-defined errors that are well outside the range of numbers used by Icon itself. Error number 500 has the predefined text program malfunction for use with runerr. This number is not used by Icon itself.

A call of runerr is subject to conversion to failure like any other run-time error.

Implementation Features: Different implementations of Icon support different features. The keyword &features provides information showing the features that are supported. It generates the name of the computer system, followed by the features of the implementation on which the current program is running. For example, for ProIcon

```
every write(&features)
```

produces

```
Macintosh
ASCII
co–expressions
error trace back
external functions
fixed regions
large integers
math functions
memory monitoring
string invocation
ProIcon extensions
```

Ordinarily, implementation features are of interest only for writing programs that are to be run on different computer systems. For example, a program that uses co-expressions can check for their presence as follows:

```
if not(&features == "co–expressions") then
   stop("co–expressions not available")
```

Memory Monitoring: Storage allocation and garbage collection are instrumented, and detailed information about these processes can be written to an *allocation history file*. See **Program Options** in Chapter 4. Appendix E describes how an allocation history file can be viewed interactively.

The function mmpause(s) causes a pause in the interactive display, giving s as the reason for the pause. The default for s is "programmed pause".

The function mmshow(x,s) redraws the display for object x as specified by s. See Appendix E for more information.

The function mmout(s) writes s as a separate line in the allocation history file.

Bugs and Limitations in Version 8

- Line numbers sometimes are wrong in diagnostic messages related to lines with continued quoted literals.

- Large-integer arithmetic is not supported in i to j and seq(). Large integers cannot be assigned to keywords.

- Large-integer literals are constructed at run-time. Consequently, they should not be used in loops where they would be constructed repeatedly.

- Conversion of a large integer to a string is quadratic in the length of the integer. Conversion of a very large integer to a string may take a very long time and give the appearance of an endless loop.

- Integer overflow on exponentiation may not be detected during execution. Such overflow may occur during type conversion.

- In some cases, trace messages may show the return of subscripted values, such as &null[2], that would be erroneous if they were dereferenced.

- Stack overflow is checked using a heuristic that may not always be effective.

- If an expression such as

```
x := create expr
```

is used in a loop, and x is not a global variable, unreferenceable co-expressions are generated by each successive create operation. These co-expressions are not garbage collected. This problem can be circumvented by making x a global variable or by assigning a value to x before the create operation, as in

```
x := &null
x := create expr
```

- Stack overflow in a co-expression may not be detected and may cause mysterious program malfunction.

10

ProIcon Extensions

ProIcon Extensions

If you've used another implementation of Icon, you should have no difficulty using ProIcon. Programs from other Icon implementations probably will run under ProIcon with little or no change. For the most part, all recent implementations of Icon have the same language facilities. Differences between implementations generally are in user interfaces and specific features of differing operating systems.

Most implementations of Icon use a command-line interface, where commands are typed in to invoke Icon, specify the program name, and so forth. ProIcon has a standard Macintosh interface. You'll do most things by selecting items from menus with a mouse. Finally, there are features that are specific to ProIcon. These include additional functions and keywords.

The following sections describe ProIcon extensions, features that are not available in the standard version of Icon. See the Mini Reference Manual (Chapter 11) for detailed information about defaults, error conditions, and so forth.

String Comparison

The value of the keyword &compare determines the method of string comparison. If &compare is 0, comparison is on a character-by-character basis. If the value of &compare is nonzero, the Macintosh international comparison method is used. See Appendix B for more information.

Function Tracing

ProIcon provides tracing for (built-in) functions as well as for (programmer-defined) procedures. Function tracing is

controlled by &ftrace. The initial value of &ftrace is 0. If the value of &ftrace is nonzero when a function is called, &ftrace is decremented and a trace message is produced in the same style as that used for procedure tracing. Function return, failure, suspension, and resumption also are traced in the style of procedures.

Suppose, for example, that hello.icn contains

```
procedure main(args)
  &ftrace := –1
  every upto(&lcase,"Hello world!")
end
```

The output is:

```
hello.icn: 3    | upto(&lcase,"Hello world!",&null,&null)
hello.icn: 3    | upto suspended 2
hello.icn: 3    | upto resumed
hello.icn: 3    | upto suspended 3
hello.icn: 3    | upto resumed
hello.icn: 3    | upto suspended 4
hello.icn: 3    | upto resumed
hello.icn: 3    | upto suspended 5
hello.icn: 3    | upto resumed
hello.icn: 3    | upto suspended 7
hello.icn: 3    | upto resumed
hello.icn: 3    | upto suspended 8
hello.icn: 3    | upto resumed
hello.icn: 3    | upto suspended 9
hello.icn: 3    | upto resumed
hello.icn: 3    | upto suspended 10
hello.icn: 3    | upto resumed
hello.icn: 3    | upto suspended 11
hello.icn: 3    | upto resumed
hello.icn: 3    | upto failed
```

Termination Dump

If the value of &dump is nonzero when an Icon program terminates, the values of all current variables are printed. The form of the dump is the same as for display(). A dump is produced regardless of whether the program terminates normally or because of an error. The initial value of &dump is 0, so no dump is produced on termination unless the value of &dump is changed.

System-Dependent Features

Some implementations of Icon have features that depend on the operating system on which Icon runs. Examples are environment variables, execution of system commands, and pipes between processes. Since the Macintosh does not have these features, they are not available in ProIcon.

ProIcon, on the other hand, has several features specifically related to the Macintosh. These are described in the following sections.

Window Functions

ProIcon provides a package of functions for manipulating windows. Windows are identified by numbers. The Interactive window is numbered 0. Window numbers increase as new windows are opened.

Newly-created windows are not displayed on the screen. You can operate on hidden windows as well as on visible ones. For example, you can write to a hidden window and then make it visible.

You can do many things with windows using the window functions. You also can connect windows to files and use ProIcon's input and output functions on them. For example, if you have a window named index data, you can open it as a file:

```
input := open("index data")
```

Then you can read it, as in:

```
while line := read(input) do
  process(line)
```

The ProIcon functions for manipulating windows are described below. All window line, column, and character positions are one-based, starting at the beginning of the window.

Opening Windows

The function wopen(name,options) opens a window with the title name according to the specified options. It returns an integer that identifies the window. If there already is a window with the title name, its window number is returned and no new window is created.

The characters in the string option determine how a new window is opened. If options contains "f", an attempt is made to read the file name into the new window. If there is no such file and "n" is contained in options, an empty window is opened. Otherwise, wopen fails.

Closing Windows

The function wclose(window,option) closes the specified window. If the window has been written to, the options determine how it is closed. If options is the empty string or null, the window contents are saved to disk in a file corresponding to the window name. If options contains an "n", the contents of the window are not saved. If options contains "a", a dialog box is presented to allow you to decide what to do. If you select Cancel in this case, the window is not closed and the function fails.

Getting window

The function wget(window,type) returns information about the specified window. The type of information desired is specified by the value of type:

0	left position of window in global coordinates
1	top position of window in global coordinates
2	width of window in pixels
3	height of window in pixels
4	position of start of selection
5	position of end of selection
6	length of text in window
7	line number of start of selection
8	character number of start of selection
9	contents of clipboard
10	selected text from window
11	horizontal coordinate of mouse in window
12	vertical coordinate of mouse in window
13	mouse button state; "" if down, fail if up
14	window number of the frontmost window
15	height of text line in pixels

Line and character positions in selections are one-based. wget fails for type = 11, 12, and 13 if the mouse is outside the specified window's text display region, which excludes the scroll and title bars.

Setting Window

The function wset(window,type,text) sets an aspect of the specified window. The aspect is specified by the value of type:

0	hide window
1	show window (with no change in front-to-back ordering)
2	show window and bring to front
3	zoom in
4	zoom out
5	discard window contents
6	move cursor left one character
7	move cursor right one character
8	move cursor to beginning of line
9	move cursor to end of line
10	move cursor up one line
11	more cursor down one line
12	move cursor and window to beginning of text (home)
13	move cursor and window to end of text
14	scroll up one page (selection not changed)
15	scroll down one page (selection not changed)
16	undo last cut / paste / clear
17	cut selection to clipboard
18	copy selection to clipboard
19	paste clipboard to selection
20	clear selection
21	copy text to clipboard
22	insert text in front of selection
23	replace selection by text

Sizing and Moving a Window

The function wsize(window,width,height) changes the size of the specified window to the given width and height, in pixels. The size of a window cannot be made smaller than what is needed for scroll bars.

The function wmove(window,left,top) moves the specified window so that its upper left corner is at the specified left and top coordinates. The corner of a window refers to its content region, which begins just below its title bar. A window may be moved off screen.

Arranging the Screen

The function warrange(method) arranges the windows on the screen according to the value of method.

0	stacked
1	tiled
2	closed

Windows in use by the input / output system and the source program window are not closed in the last case.

Screen

The value of the keyword &screen is a list containing eight values. The first four values give the x,y coordinates, in pixels, of the upper-left and lower-right corners of the main screen. The remaining four values give the coordinates of the "gray region" enclosing all screens (in the case of multiple monitors). The gray region begins below the menu bar, which is 20 pixels high.

Limiting

The function wlimit(window,lines) limits the number of lines that are retained in the specified window. If the limit is exceeded, lines are discarded from the top of the window.

Positioning

The function wgoto(window,line,character) moves the cursor in the specified window to the given line and character. If the specified line is greater than the number of lines in the window, the cursor is moved to the last line. If the specified character is greater than the number of characters on the line, the cursor is moved to the end of the line.

The function wfont(window,fontname) sets the font in the specified window. If the font does not exist, the window's font is unchanged and the function fails.

The function wfontsize(window,points) sets the font size in the specified window to the given number of points.

The function wtextwidth(window,string) returns the width of the specified string in pixels for the font and font size of the specified window.

The function wselect(window,begin,end) selects text in the specified window with the given begin and end character positions. The selected text is highlighted. Positions are one-based and start at the beginning of the window. If begin and end are the same, a vertical cursor is placed before the beginning character.

Printing a Window

The function wprint(window,setup,job) prints the contents of the specified window. If setup is nonzero, a Page Setup dialog is displayed. If job is nonzero, a Print Job dialog is displayed. If Cancel is selected from the Print Job dialog, the contents of the window are not printed and the function fails.

Other Functions

Pausing

The function delay(i) delays program execution for i milliseconds. Clock resolution is 1/60th of a second.

Dialog Boxes

The function gettext(name,okay,cancel,text) presents a dialog box with the message name. The OK button in this box has the name okay and cancel is the title for its Cancel button. The string text is presented and can be edited by the user. The value returned by the function is the edited text. The function fails if the user selects Cancel.

The function message(name,okay,cancel) is similar to gettext, except that there is no editable text.

Current Working Folder

The function currentf() returns the name of the current working folder as a full path name (with a suffix ":").

The function changef(name) changes the current working folder to name. It fails if name is not the name of a folder.

File Names

The function file(name) generates the names of the files and subfolders in folder name. It fails if name is not the name of a folder. The file names are given as partial paths (with prefix ":").

File Signatures

The function ftype(name) returns an 8-character string that consists of the type and creator signatures for the file name. If name is a folder, it returns the empty string. The function fails if name is neither a file nor a folder.

The function fset(name,type,creator) sets the type and creator signatures for name to the specified strings. If type or creator is the empty string, the corresponding signature is not changed. The function fails if name is not the name of a file.

File Dialogs

The function getfile(text,signature,name) performs a Get File Dialog for the folder name with text displayed. The string signature is used to filter file types. It may be 0, 4, 8, 12, or 16 characters long. Only files with matching signatures are displayed. The function returns the full path name of the file selected by the user. If the signature is the one-character string "f", only folders are displayed. The function returns the path name of the folder selected. This function fails if the user selects Cancel. The selected file is not opened; only its path name is returned.

The function putfile(text,prompt,name) performs a Put File Dialog in folder name with text displayed and prompt as the suggested file name. The function returns the full path name of the file selected by the user. It fails if the user selects Cancel. The selected file is not created or opened; only its path name is returned.

Launching an Application

The function launch(application,file,type) launches the named application with the specified file as its argument. If type is 0, the file is opened (as if the file were click-launched from the desktop). If type is 1, the file is printed (as if Print were selected from Finder's File menu). If MultiFinder is active, the application is sub-launched and program execution continues. The function fails if the launch fails.

External Functions

ProIcon allows programs to load and execute external functions written in other programming languages. These functions can provide services that are inefficient or impossible to perform in the Icon language. For example, specialized system calls for communications or screen graphics are accessible to external functions written in C, Pascal, or assembly language.

External functions are separately compiled code resources that are loaded dynamically as your program is executed. There are two types of external functions:

- Functions conforming to HyperCard's XCMD / XFCN interface specification (Version 1). Arguments to such functions are converted to strings, and the functions return strings. Writing such a function is relatively straightforward and requires no knowledge of Icon's internal structure.
- "Stand-alone" functions that receive data in ProIcon's internal descriptor format and return a descriptor result. Writing this type of function requires considerable knowledge of ProIcon's internal structure.

Loading and executing either form of external function is accomplished through the Icon function callout(), which specifies the type and name of the function to be loaded and the arguments supplied. The function is loaded into memory from disk if necessary and invoked with the user's arguments.

Information about writing and using external functions is contained in Appendix D.

11

Mini Reference Manual

Mini Reference Manual

This mini reference manual summarizes the built-in operations of Pro-Icon. The descriptions are brief; they're intended for quick reference only. See *The Icon Programming Language* and Chapters 9 and 10 of this manual for complete descriptions.

The operations fall into four main categories: functions, operations, keywords, and control structures. Functions, operations, and keywords perform computations, while control structures determine the order of computation. Function names provide a vocabulary used with a common syntax in which computations are performed on argument lists. Different operators, on the other hand, have different syntactic forms. They are divided into prefix operators, infix operators, and operators with different syntax. Keywords, like functions, all have common syntax, but they have no argument lists.

The descriptions in the mini reference manual are stylized. Once you become accustomed to this format, you'll be able to find things quickly.

Data types are important in Icon; you'll often need to know what types of data a function or operation expects and what type it returns. Types are indicated by letters as follows:

c	cset	C	co-expression
f	file	L	list
i	integer	N	numeric (i or r)
n	null	R	record (any record type)
p	procedure	S	set
r	real	T	table
s	string	X	any structure type (L, R, S, or T)
x	any type		

Numeric suffixes are used to distinguish different arguments of the same type. For example,

```
center(s1,i,s2)
```

indicates that center has three arguments. The first and third are strings; the second is an integer.

The type of result produced by a function follows the function, with a separating colon. For example,

```
center(s1,i,s2) : s3
```

indicates that center produces a string. The format of entries for operators and keywords is similar.

The results for generators are indicated by a sequence, as in

```
!s : s1, s2, ..., sn
```

Some operations, such as s[i], produce variables to which values can be assigned.

Icon attempts type conversion automatically when an argument does not have the expected type, so the types of arguments may be different from the expected type and still be acceptable. For example, center(s1,10,s2) and center(s1,"10",s2) produce the same result, since the string "10" is converted to the integer 10.

Default values are provided automatically in some cases when an argument is omitted (or has the null value). For example, the second argument of center defaults to 1, while the third argument defaults to a single blank. Thus, center(s1) is equivalent to center(s1,1," "). Refer to the entry for center to see how this information is shown.

Errors may occur for a variety of reasons. The possible errors and their causes are listed for each function and operation. Again, see the entry for center for examples. In particular, note that a phrase such as " s not string" means s is neither a string nor a type that can be converted to a string.

In addition to the errors listed in the entries that follow, an error also can occur if there is not enough space to convert an argument to the expected type. For example, converting a very long string to a number for use in a numerical computation conceiveably could run out of memory space. Such errors are unlikely.

Cross references among entries have two forms. Most cross references refer to functions and operations that perform related computations, such as center(), left(), and right(). There also are cross references among operators and control strcutures with similar syntax, such as x and N1 N2, even though the computations performs are not related.

Functions

The arguments of functions are evaluated from left to right. If the evaluation of an argument fails, the function is not called.

Some functions may generate a sequence of results for a given set of arguments. If an argument generates more than one value, the function may be called repeatedly with different sets of values.

abs(N) : N — compute absolute value

abs(N) produces the absolute value of N.

Error: 102 N not numeric

acos(r1) : r2 — compute arc cosine

acos(r1) produces the arc cosine of r1 in the range of 0 to π.

Errors: 102 r1 not real
205 r1 greater than 1

See also: cos()

any(c,s,i1,i2) : i3 — locate initial character

any(c,s,i1,i2) succeeds and produces i1 + 1, provided s[i1] is in c and i2 is greater than i1. It fails otherwise.

Defaults: s &subject
i1 &pos if s is defaulted, otherwise 1
i2 0

Errors: 101 i1 or i2 not integer
103 s not string
104 c not cset

See also: many() and match()

args(p) : i — get number of procedure arguments

args(p) produces the number of arguments for procedure p. For built-in procedures with a variable number of arguments, the value produced is –1. For declared procedures with a variable number of arguments, the value returned is the negative of the number of formal parameters.

Error: 106 p not procedure

See also: proc()

---

asin(r1) : r2 — compute arc sine

asin(r1) produces the arc sine of r1 in the range of $-\pi/2$ to $\pi/2$.

Errors: 102 r1 not real
205 |r1| greater than 1

See also: sin()

atan(r1,r2) : r3 — compute arc tangent

atan(r1,r2) produces the arc tangent of r1 / r2 in the range of $-\pi/2$ to $\pi/2$ with the sign of r1.

Default: r2 1.0

Error: 102 r1 or r2 not real

See also: tan()

bal(c1,c2,c3,s,i1,i2) : i3, i4, ..., in locate balanced characters

bal(c1,c2,c3,s,i1,i2) generates the sequence of integer positions in s preceding a character of c1 in s[i1:i2] that is balanced with respect to characters in c2 and c3, but fails if there is no such position.

Defaults:

	c1	&cset
	c2	'('
	c3	')'
	s	&subject
	i1	&pos if s is defaulted, otherwise 1
	i2	0

Errors:

	101	i1 or i2 not integer
	103	s not string
	104	c1, c2, or c3 not cset

See also: find() and upto()

callout(s1, s2, x1, x2, …, xn) : xm call external function

callout(s1, s2, x1, x2, …, xn) calls the external function of type s1 and name s2 with the arguments x1, x2, …, xn. The supported values of s1 are:

"XCMD"	HyperCard XCMD
"XFCN"	HyperCard XFCN
"CODE"	Stand-alone function

The maximum number of arguments for XCMDs and XFCNs is 16.

Errors:
- 216 external function not found
- 352 inadequate space for XCMD interface
- 353 could not load XCMD / XFCN

center(s1,i,s2) : s3 — center string

center(s1,i,s2) produces a string of size i in which s1 is centered, with s2 used for padding at left and right as necessary.

Defaults: i 1
s2 " "

Errors:
101 i not integer
103 s1 or s2 not string
205 i < 0
306 inadequate space in string region

See also: left() and right()

bal(c1,c2,c3,s,i1,i2) : i3, i4, ..., in locate balanced characters

bal(c1,c2,c3,s,i1,i2) generates the sequence of integer positions in s preceding a character of c1 in s[i1:i2] that is balanced with respect to characters in c2 and c3, but fails if there is no such position.

Defaults: c1 &cset
c2 '('
c3 ')'
s &subject
i1 &pos if s is defaulted, otherwise 1
i2 0

Errors: 101 i1 or i2 not integer
103 s not string
104 c1, c2, or c3 not cset

See also: find() and upto()

center(s1,i,s2) : s3 — center string

center(s1,i,s2) produces a string of size i in which s1 is centered, with s2 used for padding at left and right as necessary.

Defaults: i 1
s2 " "

Errors:
101 i not integer
103 s1 or s2 not string
205 i < 0
306 inadequate space in string region

See also: left() and right()

changef(s) : s change current working folder

changef(s) changes the current working folder to s. It fails if s is not the name of a folder.

Error: 306 inadequate space in string region

char(i) : s — produce character

char(i) produces a one-character string whose internal representation is i.

Errors:
- 101 i not integer
- 205 i not between 0 and 255, inclusive
- 306 inadequate space in string region

See also: ord()

close(f) : f — close file

close(f) closes f.

Error: 105 f not file

See also: open()

collect(i1,i2) : n — perform garbage collection

collect(i1,i2) causes a garbage collection in region i1, assuring at least i2 bytes of storage are available in that region, but fails if the requested space is not available. The regions are identified as follows:

i1	region
1	static
2	string
3	block

If i1 is 0, a collection is done, but no region is identified and i2 has no effect.

Defaults: i1 0
i2 0

Errors: 101 i1 or i2 not integer
205 i1 not between 0 and 3 inclusive or i2 < 0.

---

copy(x1) : x2 — copy value

copy(x1) produces a copy of x1 if x1 is a structure; otherwise it produces x1.

Error: 307 inadequate space in block region

cos(r1) : r2 — compute cosine

cos(r1) produces the cosine of r1 in radians.

Error: 102 r1 not real

See also: cos()

cset(x) : c — convert to cset

cset(x) produces a cset resulting from converting x, but fails if the conversion is not possible.

Error: 307 inadequate space in block region

currentf() : s — get current working folder

currentf() returns the name of the current working folder as a full path name (with suffix ":").

Error: 214 input / output error

delay(i) delays program execution for i milliseconds. Clock resolution is 1 / 60th of a second.

Default: i 0

Error: 101 i not integer

delete(X,x) : X — delete element

If X is a set, delete(X,x) deletes x from X. If X is a table, delete(X,x) deletes the element for key x from X. delete(X,x) produces X.

Error: 122 X not set or table.

See also: insert() and member()

detab(s1,i1,i2, ..., in) : s2 — remove tabs

detab(s1,i1,i2, ..., in) produces a string based on s1 in which each tab character is replaced by one or more blanks. Tab stops are at i1, i2, ..., in with additional stops obtained by repeating the last interval.

Default: i1 9

Errors:
- 101 i1, i2, ..., in not integer
- 103 s1 not string
- 210 i1, i2, ..., in not positive or in increasing sequence
- 306 inadequate space in string region

See also: entab()

display(i,f) : n — display variables

display(i,f) writes the image of the current co-expression and the values of the local variables in the current procedure call. If i is greater than 0, the local variables in the i preceding procedure calls are displayed as well. After all local variables are displayed, the values of global variables are displayed. Output is written to f.

Defaults: i &level
f &errout

Errors:
	101	i not integer
	105	f not file
	205	i < 0
	213	f not open for writing

dtor(r1) : r2 — convert degrees to radians

dtor(r1) produces the radian equivalent of r1 given in degrees.

Error: 102 r1 not real

See also: rtod()

entab(s1,i1,i2, ..., in) : s2 — insert tabs

entab(s1,i1,i2, ..., in) produces a string based on s1 in which runs of blanks are replaced by tabs. Tab stops are at i1, i2, ..., in with additional stops obtained by repeating the last interval.

Default: i1 9

Errors:
- 101 i1, i2, ..., in not integer
- 103 s1 not string
- 210 i1, i2, ..., in not positive or in increasing sequence
- 306 inadequate space in string region

See also: detab()

---

errorclear() : n — clear error indication

errorclear() clears the indications of the last error.

See also: &error

exit(i) — exit program

exit(i) terminates program execution with exit status i. If the program was run from the desktop and i is not 0, the user is left in ProIcon rather than returning to the desktop.

Default: i 0 (normal exit status)

Error: 101 i not integer

See also: stop()

exp(r1) : r2 — compute exponential

exp(r1) produces *e* raised to the power r1.

Errors: 102 r1 not real
204 overflow

See also: log() and N1 ^ N2

file(s1) : s2, s3, ..., sn — generate file names

file(s1) generates the names of the files and subfolders in the folder s1. It fails if s1 is not the name of a folder. File names are given as partial path names with a prefix ":".

Default: s1 current working folder

Errors:
	103	s1 not string
	306	inadequate space in string region

find(s1,s2,i1,i2) : i3,i4, ..., in — find string

find(s1,s2,i1,i2) generates the sequence of integer positions in s2 at which s1 occurs as a substring in s2[i1:i2], but fails if there is no such position.

Defaults: s2 &subject
i1 &pos if s2 is defaulted, otherwise 1
i2 0

Errors: 101 i1 or i2 not integer
103 s1 or s2 not string

See also: bal(), match(), and upto()

fset(s1,s2,s3) : s1 — set file signatures

fset(s1,s2,s3) sets the type and creator signatures of s1 to s2 and s3 respectively. If s2 or s3 is the empty string, the corresponding signature is not changed. fset(s1,s2,s3) fails if s1 is not the name of a file.

Defaults: s2 ""
s3 ""

Error: 306 inadequate space in string region

ftype(s1) : s2 — get file type

If s1 is a file, ftype(s1) returns an 8-character string consisting of the type and creator signatures for s1. If s1 is a folder, ftype(s1) returns the empty string. It fails if s1 is neither.

Error: 103 s1 not string
306 inadequate space in string region

get(L) : x — get value from list

get(L) produces the leftmost element of L and removes it from L, but fails if L is empty. get is a synonym for pop.

Error: 108 L not list

See also: pop(), pull(), push(), and put()

getch() : s — get character

getch() waits until a character has been entered from the keyboard and then produces the corresponding one-character string. The character is not displayed in the Interactive window. It fails on an end-of-file.

See also: getche() and kbhit()

getche() : s — get and echo character

getche() waits until a character has been entered from the keyboard and then produces the corresponding one-character string. The character is displayed in the Interactive window. It fails on an end-of-file.

See also: getch() and kbhit()

---

getfile(s1,s2,s3) : s4 — get file name

getfile(s1,s2,s3) performs a Get File Dialog for the folder s3 with s1 displayed as a user prompt. s2 is a 0-, 4-, 8-, 12-, or 16-character string that is used to filter the files displayed in the dialog. Only files with matching type signatures are displayed. getfile(s1,s2,s3) returns the the full path name of the file selected by the user. If the signature is the one-character string "f", only folders are displayed. The function returns the path name of the folder selected. The function fails if the user selects Cancel. The selected file is not opened; only its path name is returned.

Defaults: s2 no file filtering
s3 current working folder

Error: 103 s1, s2, or s3 not string
306 inadequate space in string region

See also: putfile()

---

gettext(s1,s2,s3,s4) : s5 — get text

gettext(s1,s2,s3,s4) presents a dialog box with message s1 and editable text s4. s2 is text for the OK button and s3 is the title for the Cancel button. The function returns the editable text, which may have been modified. The function fails if the user selects the Cancel button.

Defaults: s1 ""
s2 "" (OK displayed)
s3 "" (no Cancel button)
s4 ""

Error: 103 s1, s2, s3, or s4 not string

See also: message()

---

iand(i1,i2) : i3 — compute bit-wise *and*

iand(i1,i2) produces an integer consisting of the bit-wise *and* of i1 and i2.

Error: 101 i1 or i2 not integer

See also: icom(), ior(), ishift(), and ixor()

icom(i1) : i2 — compute bit-wise complement

icom(i1) produces the bit-wise complement of i1.

Error: 101 i1 not integer

See also: iand(), ior(), ishift(), and ixor()

image(x) : s — produce string image

image(x) produces a string image of x.

Error: 306 inadequate space in string region

insert(X,x1,x2) : X — insert element

If X is a table, insert(X,x1,x2) inserts key x1 with value x2 into X. If X is a set, insert(X,x1) inserts x1 into X. insert(X,x1,x2) produces X.

Default: x2 &null

Errors: 122 X not set or table
307 inadequate space in block region

See also: delete() and member()

integer(x) : i — convert to integer

integer(x) produces the integer resulting from converting x, but fails if the conversion is not possible.

See also: numeric() and real()

ior(i1,i2) : i3 — compute bit-wise inclusive *or*

ior(i1,i2) produces the bit-wise inclusive *or* of i1 and i2.

Error: 101 i1 or i2 not integer

See also: iand(), icom(), ishift(), and ixor()

ishift(i1,i2) : i3 — shift bits

ishift(i1,i2) produces the result of shifting the bits in i1 by i2 positions. Positive shift is to the left, negative to the right. Vacated bit positions are zero-filled.

Error: 101 i1 or i2 not integer

See also: iand(), icom(), ior(), and ixor()

---

ixor(i1,i2) : i3 compute bit-wise exclusive *or*

---

ixor(i1,i2) produces the bit-wise exclusive *or* of i1 and i2.

Error: 101 i1 or i2 not integer

See also: iand(), icom(), ior(), and ishift()

---

kbhit() : n — check for keyboard character

kbhit() succeeds if a character is available for getch or getche but fails otherwise.

See also: getch() and getche()

key(T) : x1, x2, ..., xn generate keys from table

key(T) generates the keys in table T.

Error: 120 T not table

launch(s1,s2,i) : s1 — launch application

launch(s1,s2,i) launches the application named s1 with argument s2. If i is 0, s2 is opened (as if s2 were click-launched from the desktop). If s2 is 1, s2 is printed (as if Print has been selected from Finder's File menu). If MultiFinder is active, s1 is sub-launched and program execution continues; otherwise, program execution terminates. launch(s1,s2,i) fails if the launch fails.

Default: i 0

Errors:
205 i not 0 or 1
306 inadequate space in string region

---

left(s1,i,s2) : s3 — position string at left

left(s1,i,s2) produces a string of size i in which s1 is positioned at the left, with s2 used for padding at the right as necessary.

Defaults: i 1
s2 " "

Errors:
101 i not integer
103 s1 or s2 not string
205 i < 0
307 inadequate space in block region

See also: center() and right()

list(i,x) : L — create list

list(i,x) produces a list of size i in which each value is x.

Defaults:

i	0
x	&null

Errors:

101	i not integer
205	i < 0
307	inadequate space in block region

log(r1,r2) : r3 — compute logarithm

log(r1,r2) produces the logarithm of r1 to the base r2.

Default: r2 *e*
x
Errors: 102 r1 or r2 not real
205 r1 <= 0 or r2 <= 1

See also: exp()

many(c,s,i1,i2) : i3 — locate many characters

many(c,s,i1,i2) succeeds and produces the position in s after the longest initial sequence of characters in c starting at s[i1]. It fails if s[i1] is not in c.

Defaults: s &subject
i1 &pos if s is defaulted, otherwise 1
i2 0

Errors: 101 i1 or i2 not integer
103 s not string
104 c not cset

See also: any()

map(s1,s2,s3) : s4 — map characters

map(s1,s2,s3) produces a string of size s1 obtained by mapping characters of s1 that occur in s2 into corresponding characters in s3.

Defaults: s2 string(&ucase)
s3 string(&lcase)

Errors: 103 s1, s2, or s3 not string
208 s2 ~= s3
306 inadequate space in string region

match(s1,s2,i1,i2) : i3 — match initial string

match(s1,s2,i1,i2) produces i1 + s1 if s1 == s2[i1+: s1], but fails otherwise.

Defaults:
- s2 — &subject
- i1 — &pos if s2 is defaulted, otherwise 1
- i2 — 0

Errors:
- 101 — i1 or i2 not integer
- 103 — s1 or s2 not string

See also: =s

member(X,x) : x — test for membership

If X is a set, member(X,x) produces x if x is a member of X but fails otherwise. If X is a table, member(X,x) produces x if x is a key of an element in X but fails otherwise.

Error: 122 X not set or table

See also: delete() and insert()

message(s1,s2,s3) : s1 — present message

message(s1,s2,s3) presents a dialog box with message s1. s2 is text for the OK button and s3 is the title for the Cancel button.

Defaults: s1 ""
s2 "" (OK displayed)
s3 "" (no Cancel button)

Error: 103 s1, s2, or s3 not string

See also: gettext()

mmout(s) : n write text to allocation history

mmout(s) writes s to the allocation history file. s is given no interpretation. Expert knowledge of allocation history files is needed to use this function.

Error: 103 s not string

See also: mmpause() and mmshow()

mmpause(s) : n — write pause to allocation history

mmpause(s) writes s to the allocation history file as a pause point with identification s.

Default: s "programmed pause"

Error: 103 s not string

See also: mmout() and mmshow()

mmshow(x, s) : n — redraw in allocation history

mmshow(x, s) specifies redrawing of x in the allocation history file. The color is defined by s as follows:

"b"	black
"g"	gray
"w"	white
"h"	highlight (also white)
"r"	normal color

If x is not in an allocated data region, mmshow() has no effect.

See also: mmout() and mmpause()

move(i) : s — move in &subect

move(i) produces &subject[&pos:&pos + i] and assigns i + &pos to &pos, but fails if i is out of range. It reverses the assignment to &pos if it is resumed.

Error: 101 i not integer

See also: tab()

name(x) : s — produce name

name(x) produces the name of the variable x. If x is an identifier or a keyword that is a variable, the name of the identifier or keyword is produced. If x is a record field reference, the record type and field name are produced with a separating period. If x is a string, the name of the string and the subscript range are shown. If x is a subscripted list or table, the type name followed by the subscripting expression is produced.

Error: 111 x not a variable

See also: variable()

---

numeric(x) : N — convert to numeric

numeric(x) produces an integer or real number resulting from converting x, but fails if the conversion is not possible.

See also: integer() and real()

open(s1,s2) : f — open file

open(s1,s2) produces a file resulting from opening s1 according to options given in s2, but fails if the file cannot be opened. The options are:

character	effect
r	open for reading
w	open for writing
a	open for writing in append mode
b	open for reading and writing
c	create
t	translate returns to linefeeds
u	do not translate returns to linefeeds

The default mode is to translate returns to linefeeds on input and to translate linefeeds to returns on output. The untranslated mode should be used when reading and writing binary files.

Default: s2 "r"

Errors: 103 s1 or s2 not string
209 invalid option

See also: close()

ord(s) : i — produce ordinal

ord(s) produces an integer (ordinal) between 0 and 255 that is the internal representation of the one-character string s.

Errors: 103 s not string
205 s not 1

See also: char()

pop(L) : x — pop from list

pop(L) produces the leftmost element of L and removes it from L, but fails if L is empty. pop is a synonym for get.

Error: 108 L not list

See also: get(), pull(), push(), and put()

pos(i1) : i2 — test scanning position

pos(i1) produces &pos if &pos = i1, but fails otherwise.

Error: 101 i1 not integer

See also: &pos and &subject

proc(x,i) : p — convert to procedure

proc(x,i) produces a procedure corresponding to the value of x, but fails if x does not correspond to a procedure. If x is the string name of an operator, i is used to distinguish between prefix and infix operators with the same symbols: i = 1 designates unary (prefix) operators, i = 2 designates binary (infix) operators, and i = 3 designates ternary (distributed) operators.

Default: i 1

Errors: 101 i not integer
205 i not 1, 2, or 3

See also: args()

---

pull(L) : x — pull from list

pull(L) produces the rightmost element of L and removes it from L, but fails if L is empty.

Error: 108 L not list

See also: get(), pop(), push(), and put()

push(L,x) : L — push onto list

push(L,x) adds x to the left end of L and produces L.

Errors: 108 L not list
307 inadequate space in block region

See also: get(), pop(), pull(), and put()

put(L,x) : L — put onto list

put(L,x) produces L and adds x to the right end of L and produces L.

Errors: 108 L not list
307 inadequate space in block region

See also: get(), pop(), pull(), and push()

putfile(s1,s2,s3) : s4 — get file name

putfile(s1,s2,s3) performs a Put File Dialog for folder s3 with s1 displayed. s2 is displayed as a user prompt in the file name box as the suggested file name. putfile(s1,s2,s3) returns the the full path name of the file specified by the user. It fails if the user selects Cancel. The selected file is not created or opened; only its path name is returned.

Default: s3 current working folder

Errors: 103 s1, s2, or s3 not string
306 inadequate space in string region

See also: getfile()

---

read(f) : s — read line

read(f) produces the next line from f, but fails on end of file.

Default: f &input

Errors:
105 f not file
212 f not open for reading
306 inadequate space in string region

See also: reads()

reads(f,i) : s — read string

reads(f,i) produces a string consisting of the next i characters from f, or the remaining characters of f if fewer remain on f, but fails on end of file. In reads(), unlike read(), returns have no special significance. reads() should be used for reading binary data.

Defaults: f &input
i 1

Errors:
101 i not integer
105 f not file
205 i <= 0
212 f not open for reading
306 inadequate space in string region

See also: read()

---

real(x) : r — convert to real

real(x) produces a real number resulting from converting x, but fails if the conversion is not possible.

Error: 307 inadequate space in block region

See also: integer() and numeric()

remove(s) removes (deletes) the file named s, but fails if s cannot be removed.

Error: 103 s not string

See also: rename()

rename(s1,s2) : n — rename file

rename(s1,s2) renames the file named s1 to be s2, but fails if the renaming cannot be done.

Error: 103 s1 or s2 not string

See also: remove()

---

repl(s1,i) : s2 replicate string

repl(s1,i) produces a string consisting of i concatenations of s1.

Errors:

	101	i not integer
	103	s1 not string
	205	i < 0
	306	inadequate space in string region

reverse(s1) : s2 — reverse string

reverse(s1) produces a string consisting of the reversal of s1.

Errors:
- 103 s1 not string
- 306 inadequate space in string region

right(s1,i,s2) : s3 — position string at right

right(s1,i,s2) produces a string of size i in which s1 is positioned at the right, with s2 used for padding at the left as necessary.

Defaults: i 1
s2 " "

Errors:
101 i not integer
103 s1 or s2 not string
205 i < 0
306 inadequate space in string region

See also: center() and left()

rtod(r1) : r2 — convert radians to degrees

rtod(r1) produces the degree equivalent of r1 given in radians.

Error: 102 r1 not real

See also: dtor()

runerr(i,x) terminate with run-time error

runerr(i,x) terminates program execution with error i and offending value x.

Default: x no offending value

seek(f,i) : f — seek to position in file

seek(f,i) seeks to position i in f but fails if the seek cannot be performed. Positions are one-based. seek(f,0) seeks to the end of the file f.

Errors: 101 i not integer
105 f not file

See also: where()

seq(i1,i2) : i3, i4, ... generate sequence of integers

seq(i1,i2) generates an endless sequence of integers starting at i1 with increments of i2.

Defaults: i1 1
i2 1

Errors: 101 i1 or i2 not integer
211 i2 is 0

See also: i1 to i2 by i3

set(L) : S — create set

set(L) produces a set whose members are the distinct values in the list L.

Default: L [ ]

Errors: 108 L not list
307 inadequate space in block region

sin(r1) : r2 compute sine

sin(r1) produces the sine of r1 in radians.

Error: 102 r1 not real

See also: asin()

sort(X,i) : L — sort list or table

sort(x,i) produces a list containing values from x. If X is a list or set, sort(x,i) produces the values of X in sorted order. If X is a table, sort(X,i) produces a list obtained by sorting the elements of X, depending on the value of i. For i = 1 or 2, the list elements are two-element lists of key/value pairs. For i = 3 or 4, the list elements are alternative keys and values. Sorting is by keys for i odd, by values for i even.

Default: i 1

Errors:
	101	i not integer
	115	X not list, set, or table
	205	i not 1, 2, 3, or 4
	307	inadequate space in block region

See also: &compare

---

sqrt(r1) : r2 — compute square root

sqrt(r1) produces the square root of r1.

Errors: 102 r1 not real
205 r1 negative

See also: N1 ^ N2

stop(x1,x2, ..., xn) — stop execution

stop(x1,x2, ..., xn) terminates program execution with an error exit status after writing x1, x2, ..., xn to the Interactive window. If the program was launched directly from the desktop, ProIcon remains active and the Interactive window remains visible.

Default: xi ""

Errors: 109 xi not string or file
213 xi file not open for writing

See also: exit()

string(x) : s — convert to string

string(x) produces a string resulting from converting x, but fails if the conversion is not possible.

Error: 306 inadequate space in string region

tab(i) : s — tab in &subject

tab(i) produces &subject[&pos:i] and assigns i to &pos, but fails if i is out of range. It reverses the assignment to &pos if it is resumed.

Error: 101 i not integer

See also: move()

table(x) : T create table

table(x) produces a table with a default value x.

Default: x &null

Error: 307 inadequate space in block region

tan(r1) : r2 — compute tangent

tan(r1) produces the tangent of r1 in radians.

Errors: 102 r1 not real
204 r1 a singular point of tangent

See also: atan()

trim(s1,c) : s2 — trim string

trim(s1,c) produces a string consisting of the characters of s1 up to the trailing characters contained in c.

Default: c ' '(blank)

Errors:
- 103 s1 not string
- 104 c not cset
- 306 inadequate space in string region

type(x) : s — produce type

type(x) produces a string corresponding to the type of x.

upto(c,s,i1,i2) : i3, i4, ... in — locate up to characters

upto(c,s,i1,i2) generates the sequence of integer positions in s preceding a character of c in s[i1:i2]. It fails if there is no such position.

Defaults: s &subject
i1 &pos if s is defaulted, otherwise 1
i2 0

Errors: 101 i1 or i2 not integer
103 s not string
104 c not cset

See also: bal() and find()

variable(s) : x — produce variable

Produces the variable for the identifier or keyword named s, but fails if there is no such variable. Local identifiers override global identifiers.

Error: 103 s not string

See also: name()

warrange(i) : i — arrange windows

warrange(i) arranges the windows on the screen according to the value of i:

i	arrangement
0	stacked
1	tiled
2	closed

Windows in use by the input / output system and the source program window are not closed in the last case.

Errors: 101 i not integer
205 i not 0, 1, or 2

See also: wopen()

wclose(i,s) : i — close window

wclose(i,s) closes window i. If the window has been written to, it is closed according to the options given by s:

s	action
""	window contents saved to disk
"n"	window contents not saved
"a"	user is queried via a dialog box

If Cancel is selected in the last case, the window is not closed and the function fails.

Default: s ""

Errors:
- 101 i not integer
- 103 s not string
- 209 invalid option
- 252 i not window identifier

See also: wopen()

---

wfont(i,s) : i — set window font

wfont(i,s) sets the font for window i to the font named s. If font s does not exist, the font is not changed and the function fails.

Errors:
- 101 i not integer
- 103 s not string
- 252 i not window identifier
- 306 inadequate space in string region

See also: wopen()

wfontsize(i1,i2) : i1 — set font size in wnindow

wfontsize(i1,i2) sets the size of the font for window i1 to i2 in points.

Errors:
- 101 i1 or i2 not integer
- 205 i2 < 1 or > 127
- 252 i1 not window identifier

See also: wopen()

wget(i1,i2) : x — get window information

wget(i1,i2) returns information about window i1 according to the value of i2:

i2	information
0	left position of window in global coordinates
1	top position of window in global coordinates
2	width of window in pixels
3	height of window in pixels
4	position of start of selection
5	position of end of selection
6	length of text in window
7	line number of start of selection
8	character number of start of selection
9	contents of clipboard
10	selected text from window
11	horizontal coordinate of mouse in window
12	vertical coordinate of mouse in window
13	mouse button state; "" if down, fail if up
14	window number of the frontmost window
15	height of text line in pixels

Line and character positions in selections are one-based. wget fails for i1 = 11, 12, and 13 if the mouse is outside the specified window's text display region, which excludes the scroll and title bars.

Errors: 101 i1 or i2 not integer
252 i1 not window identifier

See also: wopen()

---

wgoto(i1,i2,i3) : i4 — go to position in window

wgoto(i1,i2,i3) positions the cursor at line i2 and character i3 in window i1. The positions are one-based. If i2 is greater than the number of lines in the window, the cursor is set to the last line. If i3 is greater than the number of characters on the line, the cursor is set to the end of the line. The function returns the resulting character position of the cursor.

Errors: 101 i1, i2, or i3 not integer
252 i1 not window identifier

See also: wopen()

where(f) : i — produce position in file

where(f) produces the current byte position in f. Positions are one-based.

Error: 105 f not file

See also: seek()

wlimit(i1,i2) : i1 — set window output limit

wlimit(i1,i2) sets the limit for the number of lines retained in window i1 to i2. If the limit is exceeded, lines are discarded from the top of the window. If i2 is 0 or negative, there is no limit. This is relevant only if i1 is attached for output and written to in that manner.

Errors: 101 i1 or i2 not integer
252 i1 not window identifier

See also: wopen()

wmove(i1,i2,i3) : i1 — move window

wmove(i1,i2,i3) moves window i1 so that its upper-left corner is at left position i1 and top position i2 in global coordinates. Note that the corner of a window refers to its content region, which begins just below its title bar. The window may be moved off-screen.

Errors: 101 i1, i2, or i3 not integer
252 i1 not window identifier

See also: wopen()

wopen(s1,s2) : i — open window

wopen(s1,s2) opens a window with title s1 and returns an integer that identifies the window. If there is an open window on the screen with title s1, its identification is returned. Otherwise a window is opened according to the options given in s2. If s2 contains "f", an attempt is made to read file s1 into a new window. If this fails and s2 contains "n", a new window with title s1 is opened. Otherwise the function fails. The Interactive window is identified by 0 and need not be opened explicitly.

Errors:
- 103 s1 or s2 not string
- 209 invalid option
- 306 inadequate space in string region

wprint(i1,i2,i3) : i1 print window

---

wprint(i1,i2,i3) prints the contents of window i1. If i2 is nonzero, a Page Setup dialog is displayed. If i3 is nonzero, a Print Job dialog is displayed. The function fails if Cancel is selected in the Print Job dialog.

Errors: 101 i1, i2, or i3 not integer
252 i1 not window identifier

See also: wopen()

---

write(x1,x2, ..., xn) : xn — write line

write(x1,x2,..., xn) writes x1, x2, ..., xn with a return added at the end.

Default: xi ""

Errors: 109 xi not string or file
213 xi file not open for writing

See also: writes()

writes(x1,x2, ..., xn) : x — write string

writes(x1,x2, ..., xn) writes x1, x2, ..., xn without a return added at the end.

Default: xi ""

Errors: 109 xi not string or file
213 xi file not open for writing

See also: write()

wselect(i1,i2,i3) : i1 — select text in window

wselect(i1,i2,i3) selects text in window i1 starting at character i2 and ending at character i3. Character positions are one-based, starting at the beginning of the window. The selected text is highlighted. If i2 = i3, a vertical cursor is placed before character i2.

Errors: 101 i1, i2, or i3 not integer
252 i1 not window identifier

See also: wopen()

wset(i1,i2,s) : i1 — set window aspect

wset(i1,i2,s) sets an aspect of window i1 according to the value of i2:

i2	aspect
0	hide window
1	show window (with no change in front-to-back ordering)
2	show window and bring to front
3	zoom in
4	zoom out
5	discard window contents
6	move cursor left one character
7	move cursor right one character
8	move cursor to beginning of line
9	move cursor to end of line
10	move cursor up one line
11	more cursor down one line
12	move cursor and window to beginning of text (home)
13	move cursor and window to end of text
14	scroll up one page (selection not changed)
15	scroll down one page (selection not changed)
16	undo last cut / paste / clear
17	cut selection to scrap
18	copy selection to scrap
19	paste scrap to selection
20	clear selection
21	copy s to clipboard
22	insert s in front of selection
23	replace selection by s

Errors: 101 i1 or i2 not integer
252 i1 not window identifier

See also: wopen()

wsize(i1,i2,i3) : i1 — resize window

wsize(i1,i2,i3) changes the size of window i1 to width i2 and height i3, in pixels. A minimum size is enforced to allow room for scroll bars.

Errors: 101 i1, i2, or i3 not integer
252 i1 not window identifier

See also: wopen()

wtextwidth(i1,s) : i2 — compute width of text

wtextwidth(i1,s) returns the width in pixels of s using the font and fontsize for window i1. s is limited (and truncated, if necessary) to 32K characters and the width is limited to 32K pixels.

Errors:
- 101 i1 not integer
- 103 s not string
- 306 inadequate space in string region

See also: wopen()

Prefix Operations

In a prefix operation, an operator symbol appears before the operand on which it operates. If evaluation of the operand fails, the operation is not performed. If the operand generates a sequence of results, the operation may be performed several times.

There are comparatively few prefix operations. They are listed in the order of the types of operands: numeric, cset, string, co-expression, and then those that apply to operands of several different types.

+N : N — compute positive

+N produces the numeric value of N.

Errors: 102 N not integer or real
203 integer overflow

See also: N1 + N2

−N : N — compute negative

−N produces the negative of N.

Errors: 102 N not integer or real
203 integer overflow

See also: N1 − N2

~c1 : c2 — compute cset complement

~c1 produces the cset complement of c1 with respect to &cset.

Errors: 104 c1 not cset
307 inadequate space in block region

=s1 : s2 — match string in scanning

=s1 is equivalent to tab(match(s1)).

Error: 103 s1 not string

See also: match(), tab(), and N1 = N2

@C : x activate co-expression

@C produces the outcome of activating C.

Error: 118 C not co-expression

See also: x @ C

^C1 : C2 — create refreshed co-expression

^C1 produces a refreshed copy of C1.

Errors: 118 C1 not co-expression
305 inadequate space in static region

See also: N1 ^ N2

x : i compute size

x produces the size of x.

Error: 112 x not cset, string, co-expression, or a structure

See also: N1 N2

?x1 : x2 — generate random value

If x1 is an integer, ?x1 produces a number from a pseudo-random sequence. If x1 > 0, it produces an integer in range 1 to x1, inclusive. If x1 = 0, it produces a real number in range 0.0 to 1.0.

If x1 is a string, ?x1 produces a randomly selected one-character substring of x1 that is a variable if x1 is a variable.

If x1 is a list, table, or record, ?x1 produces a randomly selected element, which is a variable, from x1.

If x1 is a set, ?x1 produces a randomly selected member of x1.

Errors: 113 x1 not integer, string, or structure.
205 x1 < 0
306 inadequate space in string region if x1 string

See also: s ? *expr*

!x1 : x2, x3, ..., xn — generate values

If x1 is a file, !x1 generates the remaining lines of x1.

If x1 is a string, !x1 generates the one-character substrings of x1, and produces variables if x1 is a variable.

If x1 is a list, table, or record, !x1 generates the elements, which are variables, of x1. For lists and records, the order of generation is from the beginning to the end, but for tables it is unpredictable.

If x1 is a set, !x1 generates the members of x1 in no predictable order.

Errors:
- 103 x1 originally string, but type changed between resumptions
- 116 x1 not string, file, or structure.
- 212 x1 is file but not open for reading
- 306 inadequate space in string region if x1 string or file

/x : x — check null value

/x produces x if the value of x is the null value, but fails otherwise. It produces a variable if x is a variable.

See also: N1 / N2

\x : x — check for non-null value

\x produces x if the value of x is not the null value, but fails otherwise. It produces a variable if x is a variable.

See also: *expr* \ i

.x : x dereference variable

.x produces the value of x.

See also: R.*f*

Infix Operations

In an infix operation, an operator symbol stands between the two operands on which it operates. If evaluation of an operand fails, the operation is not performed. If an operand generates a sequence of results, the operation may be performed several times.

There are many infix operations, and finding the one you're looking for may pose a problem. They are listed first by those that perform computations (such as N1 + N2) and then by those that perform comparisons (such as N1 < N2). Assignment operations are listed last. If you can't find the operations you're looking for, try the index.

N1 + N2 : N3 — compute sum

N1 + N2 produces the sum of N1 and N2.

Errors:
- 102 N1 or N2 not integer or real
- 203 integer overflow
- 204 real overflow or underflow

See also: +N

N1 − N2 : N3 — compute difference

N1 − N2 produces the difference of N1 and N2.

Errors:
- 102 N1 or N2 not integer or real
- 203 integer overflow
- 204 real overflow or underflow

See also: −N

N1 N2 : N3 — compute product

N1 N2 produces the product of N1 and N2.

Errors:
- 102 N1 or N2 not integer or real
- 203 integer overflow
- 204 real overflow or underflow

See also: x

N1 / N2 : N3 — compute quotient

N1 / N2 produces the quotient of N1 and N2.

Errors:
- 102 N1 or N2 not integer or real
- 201 N2 = 0
- 204 real overflow or underflow

See also: /x

N1 % N2 : N3 — compute remainder

N1 % N2 produces the remainder of N1 divided by N2. The sign of the result is the sign of N1.

Errors:

	102	N1 or N2 not integer or real
	202	N2 = 0
	204	real overflow or underflow

N1 ^ N2 : N3 — compute exponential

N1 ^ N2 produces N1 raised to the power N2.

Errors:
- 102 N1 or N2 not integer or real
- 204 real overflow, underflow, or N1 = 0 and N2 <= 0
- 206 N1 < 0 and N2 real

See also: ^C, exp(), and sqrt()

x1 ++ x2 : x3 — compute cset or set union

x1 ++ x2 produces the cset or set union of x1 and x2.

Errors: 120 x1 or x2 not both cset or both set
307 inadequate space in block region

x1 – – x2 : x3 compute cset or set difference

x1 – – x2 produces the cset or set difference of x1 and x2.

Errors: 120 x1 or x2 not both cset or both set
307 inadequate space in block region

x1 x2 : x3 cset or set intersection

x1 x2 produces the cset or set intersection of x1 and x2.

Errors: 120 x1 or x2 not both cset or both set
307 inadequate space in block region

s1 || s2 : s3 — concatenate strings

s1 || s2 produces a string consisting of s1 followed by s2.

Errors: 103 s1 or s2 not string
306 inadequate space in string region

See also: L1 ||| L2

L1 ||| L2 : L3 — concatenate lists

L1 ||| L2 produces a list containing the values in L1 followed by the values in L2.

Errors: 108 L1 or L2 not list
307 inadequate space in block region

See also: s1 || s2

R.*f* : x — get field of record

R.*f* produces a variable for the *f* field of record R.

Errors: 107 R not a record type
207 R does not have field *f*

See also: .x

x1 @ C : x2 — transmit value to co-expression

x1 @ C activates C, transmitting the value of x1 to it. It produces the outcome of activating C.

Error: 118 C not co-expression

See also: @C

x1 & x2 : x2 evaluate in conjunction

x1 & x2 produces x2. It produces a variable if x2 is a variable.

N1 > N2 : N2 — compare numerically

N1 > N2 produces N2 if N1 is numerically greater than N2, but fails otherwise.

Error: 102 N1 or N2 not integer or real

See also: N1 >= N2, N1 = N2, N1 <= N2, N1 < N2, and N1 ~= N2

N1 >= N2 : N2 — compare numerically

N1 >= N2 produces N2 if N1 is numerically greater than or equal to N2, but fails otherwise.

Error: 102 N1 or N2 not integer or real

See also: N1 > N2, N1 = N2, N1 <= N2, N1 < N2, and N1 ~= N2

N1 = N2 : N2 — compare numerically

N1 = N2 produces N2 if N1 is numerically equal to N2, but fails otherwise.

Error: 102 N1 or N2 not integer or real

See also: N1 > N2, N1 >= N2, N1 <= N2, N1 < N2, N1 ~= N2, and =S

N1 < N2 : N2 — compare numerically

N1 < N2 produces N2 if N1 is numerically less than N2, but fails otherwise.

Error: 102 N1 or N2 not integer or real

See also: N1 > N2, N1 >= N2, N1 = N2, N1 <= N2, and N1 ~= N2

N1 <= N2 : N2 — compare numerically

N1 <= N2 produces N2 if N1 is numerically less than or equal to N2, but fails otherwise.

Error: 102 N1 or N2 not integer or real

See also: N1 > N2, N1 >= N2, N1 = N2, N1 < N2, and N1 ~= N2

N1 ~= N2 : N2 — compare numerically

N1 ~= N2 produces N2 if N1 is not numerically equal to N2, but fails otherwise.

Error: 102 N1 or N2 not integer or real

See also: N1 > N2, N1 >= N2, N1 = N2, N1 <= N2, and N1 < N2

s1 >> s2 : s2 — compare lexically

s1 >> s2 produces s2 if s1 is lexically greater than s2, but fails otherwise.

Errors: 103 s1 or s2 not string
306 inadequate space in string region

See also: s1 >>= s2, s1 == s2, s1 <<= s2, s1 << s2, s1 ~== s2, and &compare

s1 >>= s2 : s2 — compare lexically

s1 >>= s2 produces s2 if s1 is lexically greater than or equal to s2, but fails otherwise.

Errors: 103 s1 or s2 not string
306 inadequate space in string region

See also: s1 >> s2, s1 == s2, s1 <<= s2, s1 << s2, s1 ~== s2, and &compare

s1 == s2 : s2 — compare lexically

s1 == s2 produces s2 if s1 is lexically equal to s2, but fails otherwise.

Errors: 103 s1 or s2 not string
306 inadequate space in string region

See also: s1 >> s2, s1 >>= s2, s1 <<= s2, s1 << s2, and s1 ~== s2

s1 << s2 : s2 — compare lexically

s1 << s2 produces s2 if s1 is lexically less than s2, but fails otherwise.

Errors: 103 s1 or s2 not string
306 inadequate space in string region

See also: s1 >> s2, s1 >>= s2, s1 == s2, s1 <<= s2, s1 ~== s2, and &compare

s1 <<= s2 : s2 — compare lexically

s1 <<= s2 produces s2 if s1 is lexically less than or equal to s2, but fails otherwise.

Errors: 103 s1 or s2 not string
306 inadequate space in string region

See also: s1 >> s2, s1 >>= s2, s1 == s2, s1 << s2, s1 ~== s2, and &compare

---

s1 ~== s2 : s2 — compare lexically

s1 ~== s2 produces s2 if s1 is not lexically equal to s2, but fails otherwise.

Errors: 103 s1 or s2 not string
306 inadequate space in string region

See also: s1 >> s2, s1 >>= s2, s1 == s2, s1 <<= s2, and s1 << s2

x1 === x2 : x2 compare values

x1 === x2 produces the value of x2 if x1 and x2 have the same value, but fails otherwise.

See also: x1 ~=== x2

x1 ~=== x2 : x2 — compare values

x1 ~=== x2 produces the value of x2 if x1 and x2 do not have the same value, but fails otherwise.

See also: x1 === x2

x1 := x2 : x1 — assign value

x1 := x2 assigns the value of x2 to x1 and produces the variable x1.

Errors:
- 101 x1 requires integer, but x2 not integer
- 103 x1 requires string, but x2 not string
- 111 x1 not a variable

See also: x1 *op*= x2

x1 *op*= x2 : x1 — augmented assignment

x1 *op*= x2 performs the operation x1 *op* x2 and assigns the result to x1; it returns x1 as a variable. For example, i1 +:= i2 produces the same result as i1 := i1 + i2. There are augmented assignment operators for all infix operations except assignment operations. The error conditions for augmented assignment operations are the same as for the basic operations.

Error: 111 x1 not variable

See also: x1 := x2

---

x1 :=: x2 : x1 — swap values

x1 :=: x2 exchanges the values of x1 and x2 and produces the variable x1.

Errors:
- 101 x1 or x2 requires integer, but other argument not integer
- 103 x1 or x2 requires string, but other argument not string
- 111 x1 or x2 not a variable

See also: x1 := x2 and x1 <–> x2

x1 <– x2 : x1 — assign value reversibly

x1 <– x2 assigns the value of x2 to x1 and produces the variable x1. It reverses the assignment if it is resumed.

Errors:
- 101 x1 or x2 requires integer, but other argument not integer
- 103 x1 or x2 requires string, but other argument not string
- 111 x1 not a variable

See also: x1 := x2 and x1 <–> x2

x1 <–> x2 : x1 — swap values reversibly

x1 <–> x2 exchanges the values of x1 and x2 and produces the variable x1. It reverses the exchange if it is resumed.

Errors:
- 101 x1 or x2 is a keyword that requires an integer value, but the other argument is not an integer.
- 103 x1 or x2 is subscripted string or &subject, but other argument not string
- 111 x1 or x2 not a variable

See also: x1 <– x2 and x1 :=: x2

Other Operations

The operations on the following pages have varying types of syntax. Some have more than two operands. If evaluation of an operand fails, the operation is not performed. If an operand generates a sequence of results, the operation may be performed several times.

i1 to i2 by i3 : i1, ..., in — generate integers in sequence

i1 to i2 by i3 generates the sequence of integers from i1 to i2 in increments of i3.

Default: i3 — 1 if by clause is omitted

Errors:
- 101 — i1, i2, or i3 not integer
- 211 — i3 = 0

See also: seq()

[x1,x2, ..., xn] : L — create list

[x1,x2, ..., xn] produces a list containing the values x1, x2, ..., xn. [ ] produces an empty list.

Error: 307 inadequate space in block region

See also: list()

.Sa

x1[x2] : x3 — subscript

If x1 is a string, x1[x2] produces a one-character string consisting of character x2 of x1. x1[x2] fails if x2 is out of range. x1[x2] produces a variable if x1 is a variable.

If x1 is a list or record, x1[x2] produces element x2 of x1. x1[x2] fails if x2 is out of range.

If x1 is a table, x1[x2] produces the element corresponding to key x2 of x1.

In all cases, x2 may be nonpositive.

Errors: 101 x1 is string, list, or a record, but x2 not integer
114 x1 not string, list, table, or record.

See also: x[i1:i2], x[i1 +: i2], and x[i1 −: i2]

---

x1[i1:i2] : x2 — produce substring or list section

If x1 is a string, x1[i1:i2] produces the substring of x1 between i1 and i2. i1 and i2 may be nonpositive. x1[i1:i2] produces a variable if x1 is a variable.

If x1 is a list, x1[i1:i2] produces a list consisting of the values of x1 in the given range. In either case, it fails if out of range.

Errors: 101 i1 or i2 not integer
114 x1 not string or list
307 inadequate space in block region if x1 list

See also: x1[x2], x[i1 +: i2], and x[i1 −: i2]

.Sa

x1[i1+:i2] : x2 — produce substring or list section

If x1 is a string, x1[i1+:i2] produces the substring of x1 between i1 and i1 + i2. i1 and i2 may be nonpositive. x1[i1+:i2] produces a variable if x1 is a variable.

If x1 is a list, x1[i1+:i2] produces a list consisting of the values of x1 in the given range. In either case, it fails if out of range.

Errors:
- 101 i1 or i2 not integer
- 114 x1 not string or list
- 307 inadequate space in block region if x1 list

See also: x1[x2], x[i1:i2], and x[i1 −: i2]

---

x1[i1−:i2] : x2 — produce substring or list section

If x1 is a string, x1[i1−:i2] produces the substring of x1 between i1 and i1 − i2. i1 and i2 may be nonpositive. x1[i1−:i2] produces a variable if x1 is a variable.

If x1 is a list, x1[i1−:i2] produces a list consisting of the values of x1 in the given range. In either case, it fails if out of range.

Errors: 101 i1 or i2 not integer
114 x1 not string or list
307 inadequate space in block region if x1 list

See also: x1[x2], x[i1:i2], and x[i1 +: i2]

.Sa

x0(x1,x2, ..., xn) : xm process argument list

If x0 is a function or procedure, x0(x1,x2, ..., xn) produces the outcome of calling x0 with arguments x1, x2, ..., xn.

If x0 is an integer, produces the outcome of xi, but fails if i is out of the range 1, ..., n. In this case, it produces a variable if xi is a variable; i may be nonpositive.

Errors: 106 x0 not procedure or integer
117 x0 is main, but there is no main procedure (start up)

See also: x{...}

x0{x1,x2, ..., xn} : xm process argument list as co-expressions

---

x0{x1,x2, ..., xn} is equivalent to x0([create x1, create x2, ..., create xn]).

Error: 106 x0 not procedure or integer

See also: x(...)

---

x!L — process argument list

If x is a function or procedure, x!L produces the outcome of calling x with the arguments in the list L. If X is an integer, X!L produces L[x] but fails if x is out of range of L.

Errors: 106 x not procedure or integer
108 L not list

See also: x(…)

Keywords

Keywords are listed in alphabetical order.

Some keywords are variables; values may be assigned to these. However, the allowable type depends on the keyword. See the

&ascii : c — ASCII characters

The value of &ascii is a cset consisting of the 128 ASCII characters.

&clock : s — time of day

The value of &clock is a string consisting of the current time of day, as in "16:30:00".

&collections : i1, i2, i3, i4 — garbage collections

&collections generates the total number of garbage collections followed by the number caused by allocation in the static, string, and block regions, respectively.

&compare : i — string comparison

If the value of &compare is nonzero, string comparison is done using the Macintosh international comparison routines. If the value of &compare is zero, comparison is done by ASCII collating value. &compare is zero initially. &compare is a variable.

&cset : c all characters

The value of &cset is a cset consisting of all 256 characters.

&current : C current co-expression

The value of &current is the currently active co-expression.

&date : s date

The value of &date is the current date, as in "1989/12/31".

&dateline : s date and time of day

The value of &dateline is the current date and time of day as in "December 31, 1989 6:53 am".

&digits : c — digits

The value of &digits is a cset containing the ten digits.

&dump : i — termination dump

If the value of &dump is nonzero when an Icon program terminates, a listing of all variables and their values is produced. &dump is zero initially. &dump is a variable.

&error : i — errors to convert to failure

If the value of &error is nonzero, a runtime error is converted to expression failure and &error is decremented. &error is zero initially. &error is a variable.

&errornumber : i — number of last error

The value of &errornumber is the number of the last error converted to failure. &errornumber fails if no error has occurred.

&errortext : s — description of last error

The value of errortext is the error message corresponding to the last error converted to failure. &errortext fails if no error has occurred.

&errorvalue : x — value causing last error

The value of &errorvalue is the value that caused the last error converted to failure. &errorvalue fails if no error has occurred or no specific value caused the error.

&errout : f — standard error output

The value of &errout is standard error output. Standard error output defaults to the Intertactive window, but it may be redirected elsewhere using the Error Output ... dialog.

&fail — failure

&fail simply fails.

&features : s1, s2, ..., sn implementation features

The value of &features generates strings identifying the features of the executing version of Icon.

&file : s source file

The value of &file is the name of the file from which the current program line was compiled.

&ftrace : i function tracing

If the value of &ftrace is nonzero, a trace message is produced when a function is called, returns, suspends, is resumed, or fails. &ftrace is decremented for each message produced. &ftrace is zero initially. &ftrace is a variable.

&host : s — host system

The value of &host is a string that identifies the host system on which Icon is running. For ProIcon, this is "ProIcon for the Macintosh".

&input : f — standard input

The value of &input is standard input. Standard input defaults to the keyboard, but it may be redirected elsewhere using the Program Input ... dialog.

&letters : c — letters

The value of &letters is a cset containing the 52 upper- and lowercase letters.

&lcase : c — lowercase letters

The value of &lcase is a cset containing the 26 lowercase letters.

&level : i — procedure level

The value of &level is the integer level of the current procedure call.

&line : i source line number

The value of &line is the number of the source-program line in which it appears.

&main : C main co-expression

The value of &main is the co-expression for the main program.

&null : n null value

The value of &null is the null value.

&output : f standard output

The value of &output is standard output. Standard output defaults to the Interactive window, but it may be redirected elsewhere using the Program Output ... dialog.

&pos : i — scanning position

The value of &pos is the position of scanning in &subject. The scanning position may be changed by assignment to &pos. Such an assignment fails if it would be out of range of &subject. &pos is a variable.

&random : i — random seed

The value of &random is the seed for the pseudo-random sequence. The seed may be changed by assignment to &random. &random is zero initially. &random is a variable.

&regions : i1, i2, i3 — storage region

&region generates the current sizes of the static, string, and block regions, respectively. The size of the static region is always given as zero.

&screen : L — screen dimensions

The value of &screen is list of eight values that give the boundaries of the main screen and the "gray region" enclosing all screens. The first four values are the x,y coordinates of the upper-left and lower-right corners of the main screen, in pixels. The last four are the corresponding values for the gray region. The gray region excludes the menu bar, which is 20 pixels high.

&source : C — source co-expression

The value of &source is the co-expression for the activator of the current co-expression.

&storage : i1, i2, i3 — storage utilization

&storage generates the current amount of space used in the static, string, and block regions, respectively. The space used in the static region is always given as zero.

&subject : s — subject scanning

The value of &subject is the string being scanned. The subject of scanning may be changed by assignment to &subject. &subject is variable.

&time : i — elapsed time

The value of &time is the integer number of milliseconds since the beginning of program execution.

&trace : i — procedure tracing

If the value of &trace is nonzero, a trace message is produced when a procedure is called, returns, suspends, is resumed, or fails. &trace is decremented for each message produced. &trace is zero initially. &trace is a variable.

&ucase : c — uppercase letters

The value of &ucase is a cset containing the 26 uppercase letters.

&version : s — Icon version

The value of &version is a string identifying the version of Icon. For the current release of ProIcon, this is "ProIcon Version 2.0".

Control Structures

The way that operands of a control structure are evaluated depends on the control structure; in fact, that's what distinguishes a control structure from a function or operation.

Most control structures are identified by reserved words. They are arranged alphabetically on the following pages, with the few control structures that use operator symbols appearing at the end.

break *expr* : x — break out of loop

break *expr* exits from the enclosing loop and produces the outcome of *expr*.

Default: *expr* &null

See also: next

case *expr* of { ... } : x — select according to value

case *expr* of { ... } produces the outcome of the case clause that is selected by the value of *expr*; *expr* is limited to at most one result.

create *expr* : C — create co-expression

create *expr* produces a co-expression for *expr.*

Error: 305 inadequate space in static region

See also: ^C

every *expr1* do *expr2* — generate every result

every *expr1* do *expr2* evaluates *expr2* for each result produced by resuming *expr1*; it fails when the resumption of *expr1* does not produce a result. The do clause is optional.

fail returns from the current procedure, causing the call to fail.

See also: return and suspend

if *expr1* then *expr2* else *expr3* : x select according to outcome

if *expr1* then *expr2* else *expr3* produces the outcome of *expr2* if *expr1* succeeds, otherwise the outcome of *expr3*; *expr1* is limited to at most one result. The else clause is optional.

next — go to beginning of loop

next transfers control to the beginning of the enclosing loop.

See also: break

not *expr* : n — invert failure

not *expr* produces the null value if *expr* fails, but fails if *expr* succeeds.

repeat *expr* repeat evaluation

repeat *expr* evaluates *expr* repeatedly.

return *expr* — return from procedure

return *expr* returns from the current procedure, producing the outcome of *expr*.

Default: *expr* &null

See also: fail and suspend

suspend *expr1* do *expr2* — suspend from procedure

suspend *expr1* do *expr2* suspends from the current procedure, producing each result produced by resuming *expr1*. If resumed, *expr2* is evaluated before resuming *expr1*. The do clause is optional.

Default: *expr1* &null (only if do clause is omitted)

See also: fail and return

until *expr1* do *expr2* — loop until result

until *expr1* do *expr2* evaluates *expr2* each time *expr1* fails; it fails when *expr1* succeeds. The do clause is optional.

See also: while *expr1* do *expr2*

while *expr1* do *expr2* — loop while result

while *expr1* do *expr2* evaluates *expr2* each time *expr1* succeeds; it fails when *expr1* fails. The do clause is optional.

See also: until *expr1* do *expr2*

expr1 | *expr2* : x1, x2, ... evaluate alternatives

expr1|*expr2* generates the results for *expr1*, followed by the results for *expr2*.

See also: |*expr*

|*expr* : x1, x2, ... evaluate repeatedly

|*expr* generates the results for *expr* repeatedly, terminating if *expr* fails.

See also: *expr1* | *expr2*

expr \ i : x1, x2, ..., xi limit generator

expr \ i generates at most i results from the outcome for *expr.*

Errors: 101 i not integer
205 i < 0

See also: \x

s ? *expr* : x scan string

s ? *expr* sets &subject to s, &pos to 1, and then evaluates *expr*. The outcome is the outcome of *expr*.

Error: 103 s not string

See also: ?x

A

Character Codes

Keyboard Chart

dec.	oct.	hex.	Times	Symbol	Zapf dingbats	keyboard sequence
0	000	00				⌘-shift-option-2
1	001	01				⌘-shift-option-A; home
2	002	02				⌘-shift-option-B
3	003	03				⌘-shift-option-C; enter
4	004	04				⌘-shift-option-D; end
5	005	05				⌘-shift-option-E
6	006	06				⌘-shift-option-F
7	007	07				⌘-shift-option-G
8	010	08				⌘-shift-option-H; delete
9	011	09				⌘-shift-option-I; tab
10	012	0a				⌘-shift-option-J
11	013	0b				⌘-shift-option-K; page up
12	014	0c				⌘-shift-option-L; page down
13	015	0d				⌘-shift-option-M; return
14	016	0e				⌘-shift-option-N
15	017	0f				⌘-shift-option-O
16	020	10				⌘-shift-option-P
17	021	11				⌘-shift-option-Q
18	022	12				⌘-shift-option-R
19	023	13				⌘-shift-option-S
20	024	14				⌘-shift-option-T
21	025	15				⌘-shift-option-U
22	026	16				⌘-shift-option-V
23	027	17				⌘-shift-option-W
24	030	18				⌘-shift-option-X
25	031	19				⌘-shift-option-Y
26	032	1a				⌘-shift-option-Z
27	033	1b				⌘-shift-option-[
28	034	1c				⌘-shift-option-\; ←
29	035	1d				⌘-shift-option-]; →
30	036	1e				⌘-shift-option-6; ↑
31	037	1f				⌘-shift-option--; ↓

dec.	oct.	hex.	Times	Symbol	Zapf dingbats	keyboard sequence
32	040	20				space
33	041	21	!	!	✁	shift-1
34	042	22	"		✂	shift-'
35	043	23	#	#	✃	shift-3
36	044	24	$		✄	shift-4
37	045	25	%	%	☎	shift-5
38	046	26	&	&	✆	shift-7
39	047	27	'		✇	'
40	050	28	(	(	✈	shift-9
41	051	29	)	)	✉	shift-0
42	052	2a	*		☛	shift-8
43	053	2b	+	+	☞	shift-=
44	054	2c	,	,	✌	,
45	055	2d	-	−	✍	-
46	056	2e	.	.	✎	.
47	057	2f	/	/	✏	/
48	060	30	0	0	✐	0
49	061	31	1	1	✑	1
50	062	32	2	2	✒	2
51	063	33	3	3	✓	3
52	064	34	4	4	✔	4
53	065	35	5	5	✕	5
54	066	36	6	6	✖	6
55	067	37	7	7	✗	7
56	070	38	8	8	✘	8
57	071	39	9	9	✙	9
58	072	3a	:	:	✚	shift-;
59	073	3b	;	;	✛	;
60	074	3c	<	<	✜	shift-,
61	075	3d	=	=	✝	=
62	076	3e	>	>	✞	shift-.
63	077	3f	?	?	✟	shift-/
64	100	40	@		✠	shift-2
65	101	41	A	⌘	✡	shift-A
66	102	42	B	Β	✢	shift-B
67	103	43	C	Χ	✣	shift-C
68	104	44	D	Δ	✤	shift-D
69	105	45	E	Ε	✥	shift-E
70	106	46	F	Φ	✦	shift-F
71	107	47	G	Γ	✧	shift-G
72	110	48	H	Η	★	shift-H

dec.	oct.	hex.	Times	Symbol	Zapf dingbats	keyboard sequence
73	111	49	I	Ι	☆	shift-I
74	112	4a	J	ϑ	✪	shift-J
75	113	4b	K	Κ	✯	shift-K
76	114	4c	L	Λ	✰	shift-L
77	115	4d	M	Μ	✰	shift-M
78	116	4e	N	Ν	✭	shift-N
79	117	4f	O	Ο	✯	shift-O
80	120	50	P	Π	✮	shift-P
81	121	51	Q	Θ	✦	shift-Q
82	122	52	R	Ρ	✧	shift-R
83	123	53	S	Σ	✳	shift-S
84	124	54	T	Τ	✴	shift-T
85	125	55	U	Υ	✵	shift-U
86	126	56	V	ς	✶	shift-V
87	127	57	W	Ω	✷	shift-W
88	130	58	X	Ξ	✸	shift-X
89	131	59	Y	Ψ	✹	shift-Y
90	132	5a	Z	Ζ	✺	shift-Z
91	133	5b	[	[	✻	[
92	134	5c	\		✼	\
93	135	5d	]	]	✽	]
94	136	5e	^		✾	shift-6
95	137	5f	_	=	✿	shift--
96	140	60	`		❀	`
97	141	61	a	⌘	❁	A
98	142	62	b	β	❂	B
99	143	63	c	χ	❃	C
100	144	64	d	δ	❄	D
101	145	65	e	ε	❅	E
102	146	66	f	φ	❆	F
103	147	67	g	γ	❇	G
104	150	68	h	η	❈	H
105	151	69	i	ι	❉	I
106	152	6a	j	ϕ	❊	J
107	153	6b	k	κ	❋	K
108	154	6c	l	λ	●	L
109	155	6d	m	μ	❍	M
110	156	6e	n	ν	■	N
111	157	6f	o	ο	❑	O
112	160	70	p	π	❒	P
113	161	71	q	θ	❑	Q

dec.	oct.	hex.	Times	Symbol	Zapf dingbats	keyboard sequence
114	162	72	r	ρ	❒	R
115	163	73	s	σ	▲	S
116	164	74	t	τ	▼	T
117	165	75	u	υ	◆	U
118	166	76	v	ϖ	❖	V
119	167	77	w	ω	◗	W
120	170	78	x	ξ	❘	X
121	171	79	y	ψ	❙	Y
122	172	7a	z	ζ	❚	Z
123	173	7b	{	{	❛	shift-[
124	174	7c	|	|	❜	shift-\
125	175	7d	}	}	❝	shift-]
126	176	7e	~		❞	shift-`
127	177	7f				⌘-shift-option-/; del
128	200	80	Ä		❨	option-U, shift-A
129	201	81	Å		❩	shift-option-A
130	202	82	Ç		❪	shift-option-C
131	203	83	É		❫	option-E, shift-E
132	204	84	Ñ		❬	option-N, shift-N
133	205	85	Ö		❭	option-U, shift-O
134	206	86	Ü		❮	option-U, shift-U
135	207	87	á		❯	option-E, A
136	210	88	à		❰	option-`, A
137	211	89	â		❱	option-I, A
138	212	8a	ä		❲	option-U, A
139	213	8b	ã		❳	option-N, A
140	214	8c	å		❴	option-A
141	215	8d	ç		❵	option-C
142	216	8e	é			option-E, E
143	217	8f	è			option-`, E
144	220	90	ê			option-I, E
145	221	91	ë			option-U, E
146	222	92	í			option-E, I
147	223	93	ì			option-`, I
148	224	94	î			option-I, I
149	225	95	ï			option-U, I
150	226	96	ñ			option-N, N
151	227	97	ó			option-E, O
152	230	98	ò			option-`, O
153	231	99	ô			option-I, O
154	232	9a	ö			option-U, O

dec.	oct.	hex.	Times	Symbol	Zapf dingbats	keyboard sequence
155	233	9b	õ			option-N, O
156	234	9c	ú			option-E, U
157	235	9d	ù			option-`, U
158	236	9e	û			option-I, U
159	237	9f	ü			option-U, U
160	240	a0	†			option-T
161	241	a1	°	ϒ	❡	shift-option-8
162	242	a2	¢	′	❢	option-4
163	243	a3	£	≤	❣	option-3
164	244	a4	§	⁄	❤	option-6
165	245	a5	•	∞	❥	option-8
166	246	a6	¶	ƒ	❦	option-7
167	247	a7	ß	♣	❧	option-S
168	250	a8	®	♦	♣	option-R
169	251	a9	©		♦	option-G
170	252	aa	™	♠	♥	option-2
171	253	ab	´	↔	♠	option-E, option-E
172	254	ac	¨	←	①	option-U, option-U
173	255	ad	≠	↑	②	option-=
174	256	ae	Æ	→	③	shift-option-'
175	257	af	Ø	↓	④	shift-option-O
176	260	b0	∞	°	⑤	option-5
177	261	b1	±	±	⑥	shift-option-=
178	262	b2	≤	″	⑦	option-,
179	263	b3	≥	≥	⑧	option-.
180	264	b4	¥	×	⑨	option-Y
181	265	b5	µ		⑩	option-M
182	266	b6	∂	∂	❶	option-D
183	267	b7	∑	·	❷	option-W
184	270	b8	∏	÷	❸	shift-option-P
185	271	b9	π	≠	❹	option-P
186	272	ba	∫	≡	❺	option-B
187	273	bb	ª	≈	❻	option-9
188	274	bc	º		❼	option-0
189	275	bd	Ω	½	❽	option-Z
190	276	be	æ	¾	❾	option-'
191	277	bf	ø		❿	option-O
192	300	c0	¿	À	①	shift-option-/
193	301	c1	¡		②	option-1
194	302	c2	¬		③	option-L
195	303	c3	√		④	option-V

dec.	oct.	hex.	Times	Symbol	Zapf dingbats	keyboard sequence
196	304	c4	*f*		⑤	option-F
197	305	c5	≈		⑥	option-X
198	306	c6	Δ		⑦	option-J
199	307	c7	«	∩	⑧	option-\
200	310	c8	»		⑨	shift-option-\
201	311	c9	…		⑩	option-;
202	312	ca				option-space
203	313	cb	À		❷	option-`, shift-A
204	314	cc	Ã		❸	option-N, shift-A
205	315	cd	Õ		❹	option-N, shift-O
206	316	ce	Œ		❺	shift-option-Q
207	317	cf	œ		❻	option-Q
208	320	d0	–		❼	option--
209	321	d1	—		❽	shift-option--
210	322	d2	“	®	❾	option-[
211	323	d3	”	©	❿	shift-option-[
212	324	d4	‘	™	➔	option-]
213	325	d5	’	∏	→	shift-option-]
214	326	d6	÷	√	↔	option-/
215	327	d7	◊		↕	shift-option-V
216	330	d8	ÿ	¬	➘	option-U, Y
217	331	d9	Ÿ		➔	shift-option-`
218	332	da	⁄		➘	shift-option-1
219	333	db	¤	⇔	→	shift-option-2
220	334	dc	‹		➜	shift-option-3
221	335	dd	›		→	shift-option-4
222	336	de	fi		➞	shift-option-5
223	337	df	fl		➟	shift-option-6
224	340	e0	‡	◊	➠	shift-option-7
225	341	e1	·		➡	shift-option-9
226	342	e2	‚	â	➢	shift-option-0
227	343	e3	„	ã	➢	shift-option-W
228	344	e4	‰	ä	➤	shift-option-E
229	345	e5	Â	∑	➥	shift-option-R
230	346	e6	Ê	æ	➦	shift-option-T
231	347	e7	Á	ç	➧	shift-option-Y
232	350	e8	Ë	è	➨	shift-option-U
233	351	e9	È	é	➩	shift-option-I
234	352	ea	Í	ê	➪	shift-option-S
235	353	eb	Î	ë	➫	shift-option-D
236	354	ec	Ï	ì	➬	shift-option-F

dec.	oct.	hex.	Times	Symbol	Zapf dingbats	keyboard sequence
237	355	ed	Ì	í	⇨	shift-option-G
238	356	ee	Ó	î	⇨	shift-option-H
239	357	ef	Ô	ï	⇨	shift-option-J
240	360	f0				shift-option-K
241	361	f1	Ò		⇨	shift-option-L
242	362	f2	Ú	∫	⇨	shift-option-;
243	363	f3	Û	⌠	⇨	shift-option-Z
244	364	f4	Ù	ô	⇘	shift-option-X
245	365	f5	ı	⌡	⇢	shift-option-B
246	366	f6	ˆ	ö	⇗	shift-option-N
247	367	f7	˜	÷	⇘	shift-option-M
248	370	f8	¯	ø	⇢	shift-option-,
249	371	f9	˘	ù	⇗	shift-option-.
250	372	fa	˙	ú	→	option-H
251	373	fb	˚	û	→	option-K
252	374	fc	¸	ü	➡	⌘-shift-option-,
253	375	fd	˝	ý	➡	⌘-shift-option-'
254	376	fe	˛	þ	➢	⌘-shift-option-.
255	377	ff	ˇ	ÿ		⌘-shift-option-;

String Comparison

String (lexical) comparison is performed by the operations:

s1 >> s2	s1 greater than s2
s1 >>= s2	s1 greater than or equal to s2
s1 == s2	s1 equal to s2
s1 <<= s2	s1 less than or equal to s2
s1 << s2	s1 less than s2
s1 ~== s2	s1 not equal to s2

In each case, the operation succeeds if the specified relation holds but fails otherwise. String comparison also occurs implicitly during sorting using the function sort.

This appendix describes the string comparison, using either ProIcon's default comparison or the Macintosh international string comparison system. The selection of these two systems is controlled by &compare or by checking International Comparison in the Program Options ... dialog.

Default Comparison

ProIcon's default method of string comparison is quite fast. It is selected by setting keyword &compare to zero (the default).

Strings are compared left to right on a character-by-character basis. If the two strings are different at some position, the internal ASCII character codes of the differing characters determines the ordering of the strings. Appendix A lists all characters and their internal codes. Thus, the

string "Ac" is less than "ab" because the code for A (65) is less than the code for a (97).

If the strings are of different lengths, and all characters in the shorter string match corresponding characters in the longer string, then the shorter string is less than the longer one. Thus, "ABC" is less than "ABCD". Strings are equal only if they have the same length and all characters are the same.

While this comparison system works for most applications, the ordering implicit in the ASCII character set may be inappropriate at times. For example, under the default system, the uppercase letters collate ahead of all lowercase letters. Characters with diacritical marks, like ä, and ligatures, such as æ, are placed after all "normal" letters in the collating sequence.

This is not correct when working with languages other than English, or when attempting to produce a dictionary ordering that ignores capitalization. In these cases, the Macintosh international comparison system is useful.

International Comparison

System

The Macintosh provides support for different writing systems, or *scripts*, such as Roman, Arabic, Greek, Hebrew, and Kanji. ProIcon only supports the left-to-right Roman-based writing system.

Within the Roman-script system, Apple offers system files localized for the following languages: British, Canadian French, Danish, Dutch, Finnish, French, German, Icelandic, International English, Italian, Norwegian, Portuguese, Spanish, Swedish, Swiss French, Swiss German, Turkish, and U.S. English. These systems differ slightly in character orderings.

It is possible to create a mixed-language system. For example, a scholar might want ProIcon to use a Norwegian collating sequence on a machine with a French system file. It is also possible to create custom collating sequences. Additional information is provided at the end of this appendix.

Caveats

Prior to system version 6.0.4, the Macintosh defined its formal character set as being in the range 0 to 216 decimal, and the international comparison routines only performed special actions on those characters. Beginning with system 6.0.4, the Macintosh recognises the de facto standard that has emerged for character codes 217 to 255.

Many fonts have graphics in the range 217 to 255 (see Appendix A). Be aware that the 16 uppercase letters with diacritical marks between 217 and 244 (Ÿ, Â, Ê, Á, Ë, È, Í, Î, Ï, Ì, Ó, Ô, Ò, Ú, Û, and Ù), as well as the ligatures fi, fl, and ß participate in the international collation only if you are running system 6.0.4 or later. The remainder of this appendix is written assuming this system. If you are using system 6.0.3 or earlier, these characters do *not* sort where expected, and instead collate after all other letters strictly according to their character code values.

Going International

Setting keyword &compare to 1 or choosing International Comparison from the Program Options ... dialog tells ProIcon to invoke the Macintosh international string comparison system when making string comparisons. These comparisons are markedly slower than the default system (typically four times slower). You may wish to change the type of comparison selectively by changing the value of &compare as needed.

When international comparison is in effect, strings are compared using a system of primary and secondary orderings. Characters are assigned to classes. In each class there is a primary character used for comparisons. For example, the characters

A Á À Â Ä Ã Å a á à â ä ã å

are all assigned to one class, of which A is the primary character for comparisons. Within each class, there is a secondary, left-to-right ordering of characters.

The essence of the comparison algorithm is that strings are compared using the primary character of each class. As long as the primary characters match, the comparison proceeds. If the primary characters do not match, their relationship determines the ordering of the strings. Thus, â is less than b

because that is the ordering of their respective primary characters, A and B.

If two strings match completely in their primary ordering, the comparison routine considers secondary ordering. It remembers the first pair of characters that were different (although they yielded the same primary character), and considers their positions in their class.

For example, consider two equal-length strings that begin with Ä and À. Both produce A as their primary character. Consequently, they compare equally on the first pass, which merely considers primary characters. If all primary characters compare equally, then the comparison routine examines the secondary ordering of Ä and À. Here À appears first, and so the string containing it is less than the other.

As another example, the following tests all succeed:

```
"A" << "a"
"Ab" <<  "ab"
"ab" <<  "Ac"
```

The third case fails under the default comparison system, because the character code for a (97) is greater than the code for A (65), and the comparison never looks beyond the first character. Under the international comparison system, however, a and A are equal using their primary character, so the match proceeds, comparing b and c.

If two strings are of different lengths, the extra characters in the longer string are all considered to be greater in terms of the primary comparison. The following both succeed:

```
"a" << "Ab"
"A" << "a"
```

Ligatures provide an additional complication. These are two-letter sequences like Æ that are represented by one character in memory (code 174 in this case). For comparison purposes, the international comparison system does the following:

- Primary comparison: The ligature is expanded to the two component characters.
- Secondary comparison: The ligature is greater than the two-character sequence.

Under this system,

```
"Æd" << "AEg"
```

succeeds because Æ matches AE under their primary ordering, and the comparison is based upon d and g.

Beginning with system 6.0.4, ligatures ﬁ, ﬂ, and ß sort immediately after their expanded counterparts: fi, fl, and ss.

Character Ordering

The following table shows the primary and secondary orderings under the U.S. English Roman-writing system. Each row represents a primary class, with those toward the top of the table collating before later entries. Within a class, the secondary ordering appears from left to right. The first character in each class is its primary character. Its decimal code is also provided.

0	nul	
…	…	
31		
32	space non-breaking space (202)	
33	!	
34	" « » “ ”	
35	#	
36	$	
37	%	
38	&	
39	' ‘ ’	
40	(	
…	…	
64	@	
65	A Á À Â Ä Ã Å a á à ü â ä å ï	
66	B b ï	
67	C Ç c ç ï	letter pairs
69	E É È Ê Ë e é è ê ë ï	not
73	I Í Ì Î Ï i ı í ì î ï ý	displayed
78	N Ñ n ñ ï	are like
79	O Ó Ò Ô Ö Õ Ø o ó ï ò ô ö õ ø ï	B b
85	U Ú Ù Û Ü u ú ù û ü ï	
89	Y Ÿ y ÿ þ	
91	[	
92	\	
93	]	

94	^	
95	_	
96	‘	
123	{	
124	|	
125	}	
126	~	
127	del	
160	†	
…	…	
173	≠	
174	Æ æ Œ œ fi fl ß	(see the text for additional information about ligatures)
176	∞	
…	…	
189	Ω	
192	¿	
…	…	
198	Δ	
201	…	
208	–	
209	—	
214	÷	
215	◊	
218	⁄	
…	…	
221	›	
224	‡	
…	…	
228	‰	
240		
246	ˆ	
…	…	
251	°	
253	˝	

Other Languages

Corresponding tables for other languages differ in small ways. Here are examples of some of the differences. The list is not exhaustive, but instead gives an overview of the orderings possible with the international string comparison system.

British

The pound currency symbol (£) appears between the quotation mark class and the sharp symbol:

34	" « » “ ”
165	£
35	#

German

Characters with umlauts are expanded to the appropriate two letters, so that Ä is expanded to AE. It appears in the secondary ordering between the expanded, two-letter sequence and any equivalent ligature.

Thus the orderings are:

AE	Ä	Æ	ae	ä	æ
OE	Ö	Œ	oe	ö	œ
UE	Ü	ue	ü		

Customizing

The assembly-language programs that control the international comparison process are provided by Apple and reside within your system file. The built-in core of the comparison system is never altered. Instead, functions in the itl2 resource accommodate differences from the default actions.

The simplest method of collating under a different language is to purchase the Macintosh system software localized for that language and use it to boot your system. Done this way, the language applies system-wide.

Alternately, you can use Apple's ResEdit program to install an itl2 resource into the ProIcon application file. It should be given the same resource ID number as the existing itl2 resource in the system file. When ProIcon runs, the local

version of itl2 takes precedence and is used instead of the normal system itl2 resource. In this manner, ProIcon can be configured for one language, while the base operating system is configured for another.

itl2 resources can be obtained from other system files, or created from scratch. Creating them yourself is not a trivial task and requires considerable knowledge of Macintosh assembly-language programming. Users who wish to attempt this should obtain a copy of Macintosh Technical Note #178. It is available, free, from many public sources or from Catspaw, Inc.

C

Icon Language Checklists

Icon Language Checklists

This appendix contains several lists you may find useful when writing Icon programs.

Operator Precedence

Icon has many operators. Precedence determines how different operators, in combination, group with their operands. Associativity determines whether operations group to the left or to the right.

The list that follows gives operators by precedence from highest to lowest. Operators with the same precedence are grouped together; dotted lines separate groups. Most infix operators are left-associative. Those that associate to the right are marked as such.

It's difficult to remember all the precedences and associativities; if in doubt, use parentheses to insure that your expressions group as you expect.

(*expr*)
{*expr1*; *expr2*; ... }
[*expr1*, *expr2*, ...]
expr.f
expr1 [*expr2*]
expr1 [*expr2* : *expr3*]
expr1 [*expr2* +: *expr3*]
expr1 [*expr2* –: *expr3*]
expr1 (*expr1*, *expr2*, ...)
expr0 { *expr1*, *expr2*, ... }
.....

not *expr*
| *expr*
! *expr*
  expr
+ *expr*
– *expr*
. *expr*
/ *expr*
\ *expr*
= *expr*
? *expr*
~ *expr*
@ *expr*
^ *expr*
……
expr1 \ *expr2*
expr1 @ *expr2*
……
expr1 ^*expr2* (right associative)
……
expr1 *expr2*
expr1 / *expr2*
expr1 % *expr2*
expr1 *expr2*
……
expr1 + *expr2*
expr1 – *expr2*
expr1 ++ *expr2*
expr1 – – *expr2*
……
expr1 || *expr2*
expr1 ||| *expr2*
……
expr1 < *expr2*
expr1 <= *expr2*
expr1 = *expr2*
expr1 >= *expr2*
expr1 > *expr2*
expr1 ~= *expr2*
expr1 << *expr2*
expr1 <<= *expr2*
expr1 == *expr2*
expr1 >>= *expr2*
expr1 >> *expr2*
expr1 ~== *expr2*

expr1 === *expr2*
expr1 ~=== *expr2*
……
expr1 | *expr2*
……
expr1 to *expr2* by *expr3*
……
expr1 := *expr2* (all right associative)
expr1 <– *expr2*
expr1 :=: *expr2*
expr1 <–> *expr2*
expr1 op:= *expr2* (all augmented assignments)
……
expr1 ? *expr2*
……
expr1 & *expr2*
……
break *expr*
case *expr0* of { *expr1* : *expr2*; *expr3* : *expr4*; ... }
create *expr*
every *expr1* do *expr2*
fail
if *expr1* then *expr2* else *expr3*
next
repeat *expr*
return *expr*
suspend *expr1* do *expr2*
until *expr1* do *expr2*
while *expr1* do *expr2*

Reserved Words

The following words are reserved for use in declarations and control structures. They may not be used as identifiers.

break
by
case
create
default
do
dynamic
else
end
every

fail
global
if
initial
link
local
next
not
of
procedure
record
repeat
return
static
suspend
then
to
until
while

Generators

A few operations, called generators, may produce more than one result if the context in which they are evaluated requires it. These operations are:

bal(c1,c2,c3,s,i1,i2)
file(s)
find(s1,s2,i1,i2)
key(T)
i to j by k
seq(i1,i2)
upto(c,s,i1,i2)
!x
&collections
&features
&regions
&storage
|*expr*
expr1 | *expr2*

Run-Time Error Messages

Run-time error messages are divided into categories as indicated in the following list:

101	integer expected
102	numeric expected
103	string expected
104	cset expected
105	file expected
106	procedure or integer expected
107	record expected
108	list expected
109	string or file expected
110	string or list expected
111	variable expected
112	invalid type to size operation
113	invalid type to random operation
114	invalid type to subscript operation
115	list, set, or table expected
116	invalid type to element generator
117	missing main procedure
118	co-expression expected
119	set expected
120	cset or set expected
121	function not supported
122	set or table expected
123	invalid type
124	table expected
201	division by zero
202	remaindering by zero
203	integer overflow
204	real overflow, underflow, or division by zero
205	value out of range
206	negative first operand to real exponentiation
207	invalid field name
208	second and third arguments to map of unequal length
209	invalid second argument to open
210	non-ascending arguments to detab / entab
211	by value equal to zero
212	attempt to read file not open for reading
213	attempt to write file not open for writing
214	input / output error

215	attempt to refresh &main
216	external function not found
251	string too long for comparison
252	window identifier expected
301	evaluation stack overflow
302	system stack overflow
303	inadequate space for evaluation stack
304	inadequate space in qualifier list
305	inadequate space for static allocation
306	inadequate space in string region
307	inadequate space in block region
352	inadequate space for XCMD interface
353	could not load XCMD / XFCN
500	program malfunction

D

External Functions

External Functions

As mentioned in Chapter 10, ProIcon programs can load and execute functions written in other programming languages. These "external" functions include HyperCard XCMDs and XFCNs as well as stand-alone functions that work directly with ProIcon's internal data.

Locating External Functions

External functions exist as independent code resources residing in the resource fork of a file. Resources are identified by a four-character type string and a resource name. Some examples are:

type	resource name
"XCMD"	"Eject Disk"
"XFCN"	"Play Sound"
"CODE"	"My External Function"

When your program calls an external function, it must specify the resource type and the resource name. ProIcon then searches the resource forks of all open files for the appropriate resource. The search begins with the resource fork of your executable Icon file. If it is not found there, the resource fork of the ProIcon application is examined. If it still is not found, your System file is searched.

Your program can open additional files for resources prior to invoking an external function. Such files move to the head of the list of open files, and are searched before all others. Two external functions that open and close files for resources are present in ProIcon, making them always accessible. To open a file for resources, use:

```
resFile := callout("CODE", "OpenResFile", name)
```

where *name* is the string name of the file.

This call fails if the file cannot be opened. The file is closed with:

```
callout("CODE", "CloseResFile", resFile)
```

Managing Resources

Code and other resources can be copied from file to file using Apple's ResEdit program. Alternately, development tools such as LightspeedC have the ability to build code resources and merge them into the resource fork of an existing file.

When copying code resources, particularly XCMDs and XFCNs, also be sure to copy any other resources present and required by the code resource (such as dialogs, dialog items, and strings).

Compiling and running Icon programs is a two-stage process. Icon source files are compiled to produce intermediate files. One or more intermediate files are then linked to produce an executable file. In parallel with this process, ProIcon copies and merges resources found in the resource forks of the source and intermediate files.

Under this system, you can install your external function and any associated resources in the resource fork of your Icon source file. When ProIcon compiles the program, these resources are copied to the intermediate file, which has the suffix .u1. During the linking phase, resources from all linked files are merged into the executable file. Duplicated resources produce this fatal error message:

```
Resource duplicates a previously copied resource:
Type n, ID m, name s
```

The executable file's resource fork is automatically opened during execution, making these resources available to your program.

If you develop an extensive collection of external functions, this system encourages you to group them into small intermediate files that are linked by your main source file. These files can contain small Icon procedures that serve as Icon-language "wrappers" for the external functions. Alternatively, source files can be empty, or just contain comment lines, but have resources in their resource forks. To create such an intermediate file, simply uncheck the Link Files menu item in the Run menu and compile the source file.

Global and Static Data

Macintosh applications reference their global and static data using machine register A5. External functions are stand-alone code resources, and

have no direct access to the host application's global variables. However, an external function can maintain its own private global and static variables to communicate among the various functions and routines that comprise the external function. If you are using assembly language or a Symantec compiler, this is a simple operation. If you are using MPW tools, creating a private global and static environment is more involved.

Using Symantec Compilers

When constructing code resources, private global and static data are appended to the end of the code resource. Stand-alone code resources are compiled to access this data as offsets from register A4, which points to the start of the code resource.

ProIcon sets up register A4 properly prior to calling your external function, so no special action is required. However, if you are writing an XCMD or XFCN that will be used with other systems, you should set up and restore register A4 when entering and leaving your function. For example, using LightspeedC, upon function entry execute:

```
RememberA0();       / save code resource address found in A0 /
SetUpA4();          / save entry A4, load with saved A0 value /
```

Just prior to returning from the function, execute:

```
RestoreA4();
```

Remember, if you are creating external functions other than XCMDs and XFCNs, there is no need to use this code, because ProIcon takes care of things for you.

Global and static data placed within the code resource are not guaranteed to persist between function calls because the external function could be purged from memory and then reloaded. If persistence is required, mark the code resource non-purgeable. You can do this when you create it ("set project type...", "attrs" in LightspeedC), or afterwards by using ResEdit.

Code resources are always purged from memory when the program terminates because ProIcon closes all open resource files other than itself. To maintain persistence across runs, install the resource in the ProIcon application proper.

Using Assembly Language

If you are writing in assembly language, use the method described above for Symantec compilers. That is, place your global and static data at the end of your code resource, and reference it as a relative offset from the start of the code resource. Upon function entry, registers A0 and A4 point to the

start of the code resource, and they can be used as base pointers for addressing.

Using MPW Compilers

Code resources constructed using MPW tools expect register A5 to point to the global and static variables. But register A5 points to ProIcon's global variables, and these cannot be used by the stand-alone code resource. To use global variables and static data, these steps must be taken:

1. Save ProIcon's register A5.
2. Allocate memory for the global and static data (the "A5 world").
3. Initialize the memory (C allows globals and static variables to have initial values).
4. Deallocate the memory and restore register A5 prior to returning.

The methods for doing these things are non-trivial and beyond the scope of this document. Consult the August, 1990 revision of Macintosh Technical Note 256, "Stand-Alone Code" for more information.

Accessing QuickDraw

An external function often needs to access the QuickDraw globals of the ProIcon application for which it is performing a service. QuickDraw's drawing operations assume a properly-initialized QuickDraw world, which is provided by ProIcon. Most QuickDraw calls are supported and no special effort is required. One limitation, however, is that explicit references to QuickDraw globals like thePort and screenBits are not allowed. A linker cannot resolve the offsets to these variables because it does not process a stand-alone module along with ProIcon.

Since the structure of the QuickDraw global data is known, and there is a pointer to it in the A5 world, stand-alone code can reference any desired QuickDraw global indirectly. The following QuickDraw variables are affected: thePort, white, black, gray, ltGray, dkGray, arrow, screenBits, and randSeed.

The following C code shows how stand-alone code can obtain a pointer to the QuickDraw globals. Using this pointer, the globals can be referenced via a structure reference.

```
struct QDVarRec {
  long      randSeed;
  BitMap    screenBits;
  Cursor    arrow;
  Pattern   dkGray;
```

```
    Pattern    ltGray;
    Pattern    gray;
    Pattern    black;
    Pattern    white;
    GrafPtr    thePort;
    }  pQDGlobals;                    /  Pointer to QuickDraw Globals  /

   /
     Develop pointer to Quick Draw Globals
    /
    pQDGlobals = (struct QDVarRec  )(  (long  )CurrentA5 –
              (sizeof(struct QDVarRec) – sizeof(thePort)));
```

Given this pointer in pQDGlobals, reference the needed variables as follows: pQDGlobals–>thePort, pQDGlobals–>screenBits, etc.

For additional information, see Macintosh Technical Note 256, "Stand-Alone Code".

Interfacing XCMDs and XFCNs

HyperCard scripts can invoke external commands (XCMDs) and external functions (XFCNs). Although HyperCard uses different syntaxes to invoke the two, ProIcon makes no such distinction. Both accept string arguments (or arguments that can be converted to strings), and return a string result.

ProIcon loads and calls XCMDs and XFCNs via Icon's function callout(), which takes the form:

```
callout(type, resource_name, arg1, arg2, …, arg16)
```

where:

type	is the 4-character string "XCMD" or "XFCN"
resource_name	is a string used to specify the function's resource by name
arg1 … arg16	are optional arguments to be converted to strings and passed to the function. The HyperCard interface imposes a limit of 16 arguments.

callout() succeeds and returns a string result (which may be the empty string). There is no way for an XCMD or XFCN to signal failure or to suspend. ProIcon produces an error message if the external function cannot

be found, if insufficient memory is available to convert arguments or return the result string, or if the XCMD or XFCN sets the passFlag true in the XCmdBlock record structure (described below).

After loading the external function, callout() performs a Pascal-style subroutine call to the first memory location of the external function. A single parameter is on the stack: a pointer to an XCmdBlock record structure. This structure contains a count of the number of arguments, an array of handles pointing to null-terminated strings for the input arguments, and a place to return a handle to a null-terminated result string. The C-language form of the XCmdBlock structure is contained in file IconXCmd.h on the distribution disk in folder External Functions:XCMDs/XFCNs:C Sources.

The XCmdBlock structure also defines a "callback" mechanism by which external functions can obtain access to 29 different services provided by HyperCard. Because HyperCard is not present, ProIcon emulates these callbacks. For 20 of these callbacks, the emulation exactly mirrors HyperCard's behavior. The remaining nine callbacks are HyperCard-specific, and fail gracefully by returning an xresFail result code.

The 20 callbacks that are fully implemented are:

BoolToStr	Boolean to true / false string conversion
ExtToStr	extended (80-bit) real-to-string conversion
GetGlobal	get global value
LongToStr	unsigned long-to-string conversion
NumToHex	long-to-hexadecimal conversion
NumToStr	signed-long-to-string conversion
PasToZero	Pascal-string-to-C string with handle
ReturnToPas	copy up to return character or end of C string
ScanToReturn	scan to return character or end of C string
ScanToZero	scan to end of C string
SetGlobal	set global value
StringEqual	case-insensitive string-equal compare
StringLength	C string length
StringMatch	case-insensitive string search
StrToBool	true / false string to Boolean conversion
StrToExt	string-to-extended real (80-bit) conversion
StrToLong	string-to-unsigned-long conversion
StrToNum	string-to-signed-long conversion
ZeroBytes	zero bytes in memory
ZeroToPas	C-string-to-Pascal-string

The following nine callbacks are HyperCard specific, and are not emulated. All return xresFail (integer 1) as a result code. In addition, the

GetFieldBy… callbacks return a handle to an empty string, mimicking HyperCard's behavior when presented with an unrecognized field name.

EvalExpr	evaluate HyperTalk expression
GetFieldByName	get contents of card field by name
GetFieldByNum	get contents of card field by number
GetFieldByID	get contents of card field by ID
SendCardMessage	send message to card
SendHCMessage	send message directly to HyperCard
SetFieldByName	set card field contents by name
SetFieldByNum	set card field contents by number
SetFieldByID	set card field contents by ID

XCMDs and XFCNs return a result string by placing a handle to a null-terminated string in the returnValue field of the XCmdBlock. ProIcon disposes of the handle after copying the result string. The string is returned as the value of the callout function.

A complete description of XCMD and XFCN programming is beyond the scope of this manual. The interested reader should consult books specifically written on the topic. See the references in Appendix F.

Note that XCMDs and XFCNs pass data as handles to strings. These strings must be converted from ProIcon's internal format to null-terminated strings in the application heap, making the interface somewhat inefficient. However, the interface does provide access to a wide range of public-domain and commercial XCMDs and XFCNs.

XCMDs and XFCNs that rely upon the nine HyperCard-specific callbacks should not be used with ProIcon. Doing so can lead to unpredictable results, and may lock up your computer if the function does not examine the callback result code for errors. Fortunately, there is an ever-growing number of XCMDs and XFCNs that anticipate that they may not be called from HyperCard.

XCMD and XFCN Globals

Note that the callbacks that get and set global values refer to globals maintained by ProIcon's XCMD / XFCN interface. They do not refer to global variables in your Icon program. They provide a method by which XCMDs and XFCNs can save persistent data between function calls. New globals are created with the SetGlobal callback. A GetGlobal reference to an undefined global returns a handle to an empty string. All globals are deleted when an Icon program terminates — there is no persistence between program runs. Global names are not case sensitive.

Some XCMDs and XFCNs may require the user to set up certain global values prior to calling the external function. This can be done from your Icon program using the GetGlobal and SetGlobal procedures provided in Globals.icn, located in folder External Functions:XCMDs/XFCNs.

```
link Globals
    …
    SetGlobal(s1, s2)
```

where s1 is a string containing the global name, and s2 is a string containing the global value. Each string must be less than 256 characters in length.

XCMD / XFCN globals can be copied into an Icon program with:

```
gvalue := GetGlobal(s)
```

where s is the global name.

Interfacing Other Code Resources

Code resources other than XCMDs and XFCNs can be used as external functions. Access is through the callout() function, but the string conversion of arguments required by the XCMD / XFCN interface is not done. Arguments are delivered in Icon's internal formats, and the function result is constructed in Icon's string or block region.

ProIcon recognizes this style of function call when the resource type string is not "XCMD" or "XFCN". The resource is identified with a resource name string. The methods of locating and managing resources described earlier in this appendix remain the same.

Because there is no data conversion, this form of external function is more efficient than the XCMD and XFCN forms. The drawback is that you must have some knowledge of ProIcon's internal organization to write functions that behave properly.

ProIcon's internal structures mirror the structures described in *The Implementation of the Icon Programming Language* (see Appendix F). A full discussion of these structures is beyond the scope of this manual. However, some basic concepts and simple data types such as strings, integers, and real numbers are described here. Together with the sample programs provided in the folder External Functions:Samples, you should be able to construct useful functions.

Implementation Fundamentals

Arguments are passed to a function as an array of *descriptors* in memory. Each descriptor consists of two long (4-byte) words: a type word and a

value word. The type word identifies the type of data represented and the value word contains either a pointer to the data, or the data itself. Flag bits in the type word provide additional information.

In the following discussion, you may wish to have a copy of the file ProIcon.h available for reference (found in the ExternalFunctions:Samples folder).

Argument descriptors are in one of four forms, depending upon the data represented:

The null value: The type word contains the T_Null type code, and the F_Nqual flag bit. The value word is zero.

Strings: String descriptors are called *qualifiers*. The type word contains the length of the string. The value word points to the first character of the string. There are no flag bits set in the type word, and the string is *not* null terminated. The absence of the F-Nqual flag bit distinguishes this qualifier from all other descriptors and indicates that the type field contains a length, not a type code.

Integers: The type word contains the value D_Integer, which is a combination of the T_Integer type code and the F_Nqual flag bit. The value word contains the integer.

Reals and all other data types: The type word contains the type code, and the F_Ptr and F_Nqual flag bits. Other flag bits may be present as well. The value word points to a block of memory that contains the actual data. The size and layout of this block depends on the data type, although all blocks begin with a title word containing the type code.

The file ProIcon.h contains structure declarations for each block type. For example, the b_real structure shows that a real value is stored in a block consisting of the title word followed by the real number.

Function results are returned by creating a descriptor for the result value. For results other than integers and the null value, the descriptor's value word contains a pointer to a string or data block. The string or data block must be allocated in ProIcon's string or block region respectively. A callback mechanism is provided to accomplish this.

Function Skeleton

External functions are called with a Pascal-style subroutine jump to the first location of the external function. "Pascal-style" means that arguments are pushed left-to-right and the external function removes its calling arguments from the stack. Begin your external function with this sequence (shown here in LightspeedC):

```
#include "ProIcon.h"

pascal dptr main(dargv, argc, ip, callback)
struct descrip dargv[];
short int argc;
short int  ip;
pointer (  callback)();
{
```

This skeleton specifies that main() is a Pascal-style function returning a pointer to a descriptor (dptr). It takes four arguments:

dargv is a pointer to an array of descriptors containing the Icon arguments to the callout function. Consider this call:

```
callout("CODE", "myResourceName", 123, "test string", 54.9)
```

The descriptors in the dargv array contain the following:

dargv[0]	descriptor used to return result
dargv[1]	the integer 123
dargv[2]	the string "test string"
dargv[3]	the real number 54.9.

Notice that the first "true" user argument always is dargv[1].

argc is a count of the number of user arguments in the dargv array. In this example, its value is 3.

Arg0, Arg1, …, Arg6 are definitions in ProIcon.h for dargv[0], dargv[1], …, dargv[6], respectively.

ip is a pointer to a short integer that is used to signal error conditions. Its usage is described later in this appendix.

Finally, callback is the entry point of a C function within ProIcon that can be called to provide utility functions. These callback routines are readily accessed through the definitions provided in ProIcon.h.

Allocating Memory

Returning a result other than an integer or null value requires the allocation of memory. This is a two-step process:

1. A request for memory is made specifying the number of bytes needed. No memory is allocated at this time, but a garbage collection may occur to assure that the requested amount of space is available when it is needed. All references to Icon data must be in argument descriptors at this time. Specifically, pointers to Icon string or block data may not be in C variables, since the data may

move but C variables are not changed. If a garbage collection occurs, pointers in the input argument list are adjusted accordingly.

2. When storage is actually needed, an allocation request is made to actually allocate memory in the string or block region.

If an external function performs several allocations to build a composite result (such as a list), it must perform a request for *all* memory needed prior to doing any of the actual allocations.

For examples of functions allocating memory for specific data types (csets, external blocks, large integers, reals, and strings), examine the programs in the folder External Functions:Samples. Note that in the case of real values, the memory request and allocation steps have been combined within one function, makereal().

External Data Blocks

An external function may allocate external data blocks. These are memory blocks of arbitrary size whose contents are entirely up to the external function. ProIcon never looks at the contents of the external block. You should not store a relocatable pointer (a pointer to a block or string within ProIcon's block or string region) within an external block, because it will not be adjusted by ProIcon during a garbage collection.

External blocks can be assigned to an Icon variable or stored in an element of an Icon structure. They may be duplicated with the Icon copy() function. Since external blocks reside in the block region, they may be moved as the result of a garbage collection. Like other Icon objects, their space is released when it is no longer referenced.

Returning Results

The external function returns a pointer to a descriptor containing the result value. The descriptor present at Arg0 (dargv[0]) should be used for this purpose. After building a result descriptor in Arg0, the function should return by executing

```
return &Arg0;
```

The Return macro provided in ProIcon.h performs this return.

Signaling Errors

External functions are provided with a pointer to a short integer (argument ip) that can be used to signal an error condition. The integer is initialized to –1 prior to each call, signaling "no error". Setting it to a positive value triggers ProIcon's runtime error mechanism. A list of supported error numbers is contained in the ProIcon.h file.

In addition to setting an error number, the function still has the option of returning either a pointer to a descriptor or a null pointer. If a descriptor is returned, the value it contains is displayed after the error message. This is most useful for providing diagnostics about improper input arguments.

ProIcon.h contains a macro named RunErr() that encapsulates the process. For example,

```
RunErr(Err101, &Arg1);
```

displays error number 101 ("integer expected") and displays the first true argument to the function.

Definitions, Macros, and Typedefs

Numerous definitions, macros, and typedefs are provided in ProIcon.h to allow you to write functions more conveniently. A list of the more important ones is provided here; consult the ProIcon.h file for a complete list and further information:

Typedefs

word	the basic ProIcon memory unit; a 32-bit integer
uword	unsigned word
dptr	pointer to a descriptor
pointer	generic pointer

Definitions

Arg0	descriptor for result of function
Arg1	descriptor of first argument (dargv[1])
…	…
Arg6	descriptor of sixth argument (dargv[6])
F_Ptr	flag bit in descriptor type word if the value word is a pointer
MaxCvtLen	maximum string length in conversions
T_*n*	type code for data type *n*
D_*n*	type code plus flag bits for data type *n*
Err*nnn*	definition for error number *nnn*
CvtFail	attempted conversion failed
Cvt	attempted conversion succeeded
NoCvt	attempted conversion wasn't necessary (already in correct form)
Emptydp	pointer to descriptor for the empty string
Nulldp	pointer to descriptor for the null value
Onedp	pointer to descriptor containing integer one

Zerodp	pointer to descriptor containing integer zero
Error	error signal
Failure	failure signal
Return	return from function with value in Arg0
Success	success signal

Macros

BlkLoc(d)	block pointer in descriptor d
BlkType(x)	type of block pointed at by x
ChkNull(d)	test for null-valued descriptor
EqlDesc(d1,d2)	test for two descriptors being equivalent
Fail	return from function signaling failure
IntVal(d)	integer in the value field of a descriptor
MakeInt(i,dp)	build an integer in descriptor pointed to by dp
Pointer(d)	test if descriptor d contains a pointer
RealVal(d)	real number in block specified by descriptor d
RunErr(n,dp)	return error number n with value in descriptor pointed to by dp
Qual(d)	test if descriptor d is a qualifier (refers to a string)
StrLen(q)	length of string specified by qualifier q
StrLoc(q)	location of first character of string
Type(d)	type code of descriptor

Callback Functions and Values

The callback argument provided to your function offers a way to obtain values and services from ProIcon. A complete list of values and services provided is found in ProIcon.h.

The vcallback function provides the version number of the callback interface, which currently is 1. This number is incremented in each ProIcon release that adds new callbacks. Before using these new callbacks, an external function should test that it is executing with a version of ProIcon that provides these new functions.

The following list shows the more common callbacks:

vcallback	return a word value providing the version number of the interface
alccset()	allocate a cset block
alcextrnl(n)	allocate an external data block of n words
alcreal(val)	allocate real block and store val in it
alcstr(s,slen)	allocate string space of size slen characters, and store string s in it

blkreq(n)	request n bytes will be needed in the block region. Returns Success or Error.
cvint(dp)	convert descriptor pointed to by dp to an integer. Returns T_Integer or CvtFail.
cvreal(dp)	convert descriptor pointed to by dp to a real number. Returns T_Real or CvtFail.
cvstr(dp, sbuf)	convert data in descriptor pointed to by dp into a string, using sbuf (of size MaxCvtLen) as a buffer if necessary. Returns CvtFail if the conversion fails, Cvt if dp was not a string but was converted to one and the result is in sbuf (in which case it is null-terminated), and NoCvt if dp was a string.
makereal(r, dp)	allocate real block to hold real value r. Descriptor at dp will point to block.
qtos(dp, sbuf)	convert a string pointed to by dp to a C-style string. Put the C-style string in sbuf if it will fit, otherwise put it in the string region. sbuf must be MaxCvtLen bytes long. Returns Success or Error if inadequate memory is available.
strreq(n)	request n bytes are needed in the string region. Returns Success or Error.

Conclusion

External functions provide a mechanism to execute programs written in other programming languages. Most users should consider the XCMD and XFCN form of external functions because it does not require knowledge of ProIcon's internal organization.

For persons with more demanding requirements, the raw descriptor interface offered provides more power at the cost of greater complexity. Incorrectly formed data structures are likely to result in system crashes at times far removed from the original function call. A useful debugging technique in this regard is to place collect(0,0) function calls on either side of your external function. The garbage collection may discover problems caused by your function before they are allowed to propagate further through the system.

The external function interface is an evolving one. Look for a README file in the External Functions folder for new features and information that became available after this manual was published.

E

Memory Monitoring

Memory Monitoring

Icon has a large repertoire of data types, including strings, csets, lists, sets, tables, and records. These data types and operations on them provide much of the richness of the language and make it possible to represent and process complex data with relative ease.

Storage Management

Behind the scenes, as your program runs, Icon manages data, allocating space as it is needed, and collecting unused data that is no longer in use (garbage) when space runs out.

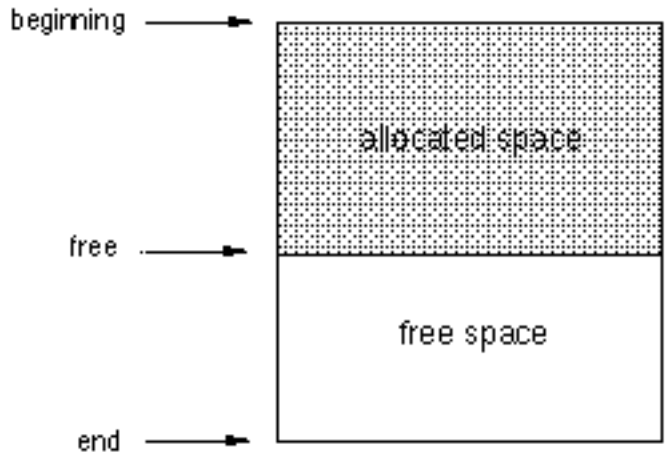
Such complex storage management operations are, of course, far removed from the simple computational operations of a programming language like Fortran, in which most operations closely mimic the instructions of the computer on which it runs. In fact, the vast difference between the language operations of Icon and the machine instructions that ultimately carry them out is what makes Icon so useful for complicated problems.

Although you generally aren't aware of storage management when you write or run an Icon program, storage management often is important, since memory is a scarce commodity and storage management can account for a significant amount of the time spent executing some programs. And what goes on behind the scenes is interesting, even fascinating.

Icon allocates space for objects that are created during program execution in two regions: a string region and a

block region. The string region consists of characters, while the block region contains structures and related objects. Allocation in the string region is in terms of bytes, while allocation in the block region is in terms of 4-byte “words”.

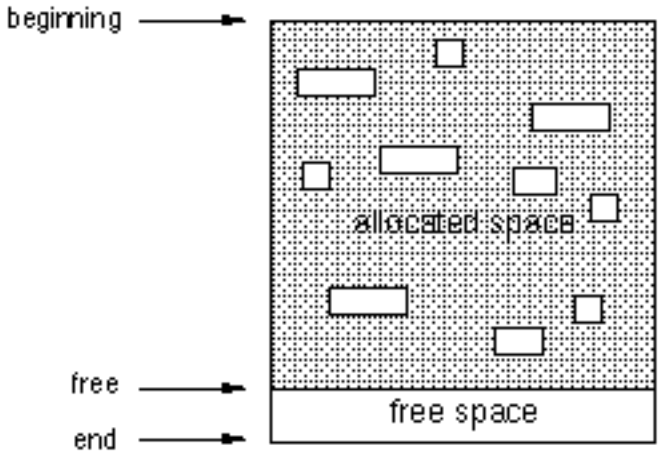
Allocation proceeds in the same manner in both main regions. The regions initially are empty and bounded by pointers. As space is needed, it is provided starting at the



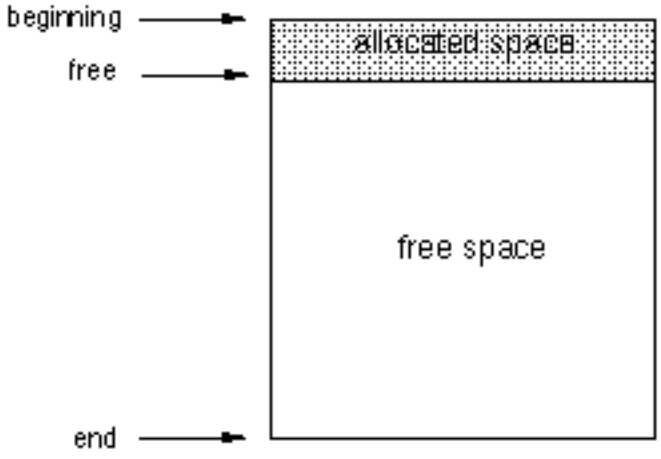
beginning of the region. A “free” pointer is incremented to mark the boundary between *allocated space* and *free space*.

If there is not enough available free space to satisfy an allocation request, a garbage collection is performed to reclaim space occupied by objects that are no longer needed.

The garbage collection process is fairly complicated, since it is necessary to locate all objects that may be



needed for subsequent program execution. Objects that need to be saved typically are scattered throughout the string and block regions:



Once the objects to be saved are identified (“marked”), they are relocated toward the beginning of their region, compressing the allocated space and making more free space available so that allocation can proceed:

Allocation History Files

The implementation of ProIcon is instrumented so that the details of storage management can be recorded. When this instrumentation is enabled, it writes an *allocation history file* that contains a record of every data object that is allocated during program execution, as well as the details of garbage collection.

Although an allocation history file contains all the details of storage management during the execution of a program, it is virtually incomprehensible — simply because it is so detailed.

ProIcon provides a tool, called MemMon, that analyzes allocation history files and presents Icon's allocated data regions in color (or in black-and white on a gray-scale monitor). A monitor with at least a four-bit color or gray-scale video card is required, and an eight-bit card is preferred. MemMon will not run on a one-bit black and white system.

Each different type of data is shown in a different color for easy recognition. MemMon animates the allocation process, showing each object as it is allocated, how much space it occupies, and where it is in memory. When a garbage collection occurs, the process is shown in detail, with the objects that are collected and those that are saved clearly identified. The display can be stopped at any time and PICT snapshots can be saved for future use.

Such displays can show the relative cost of using various types of data, help locate unnecessary storage allocation, and in general give you a "feeling" for what your program is doing.

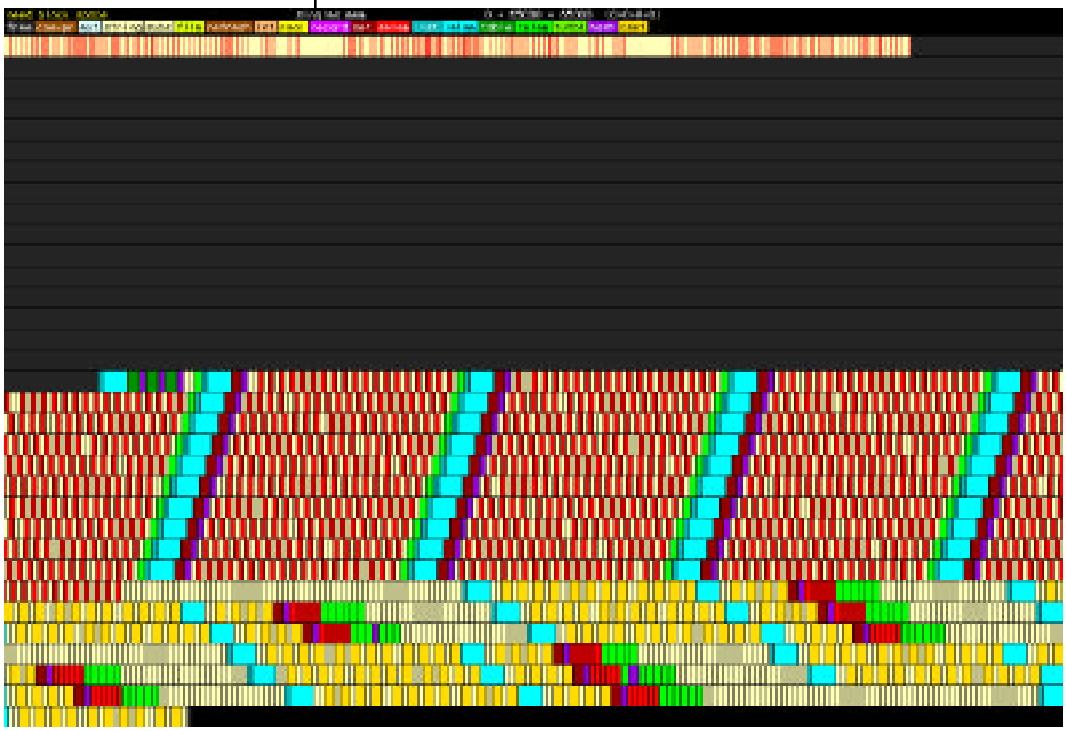
In order to use MemMon to display the details of storage management, you first need to create an allocation history file. To do this, select Program Options ... from the Options menu before running your program. In the resulting menu, check Memory Monitoring as described in Chapter 4. When you run your program, an Open dialog will appear to let you specify the name of a file to receive the allocation history information. Once you've done that, your program executes as it normally would and the allocation history file is written.

Enabling memory monitoring does not affect the computations performed by your program, but it does slow it down somewhat and allocation history files can be quite large for programs that allocate a lot of data and that garbage collect frequently. Consequently, memory monitoring should only be enabled when you want an allocation history file.

MemMon is separate from ProIcon; you must leave the ProIcon application to view an allocation history file.

Memory Displays

Before explaining how to run MemMon, it will be helpful to know what the displays it produces look like and what they mean. Basically, they are refined versions of the schematic representations of memory shown on the preceding pages. Here's an example:



At the top of the display is a two-line legend. The first line shows the program state at the left, the name of the allocation history file in the center, and storage information at the right. The storage information gives the

region sizes first with separating plus signs. The four values in parentheses are the number of garbage collections triggered by allocation in each of these regions, followed by the number of garbage collections caused by explicit calls of the function collect().

The second line of the legend lists the names of all the allocated types. Some are abbreviated because of the limited space available. The words on the display's second legend line have the following meanings:

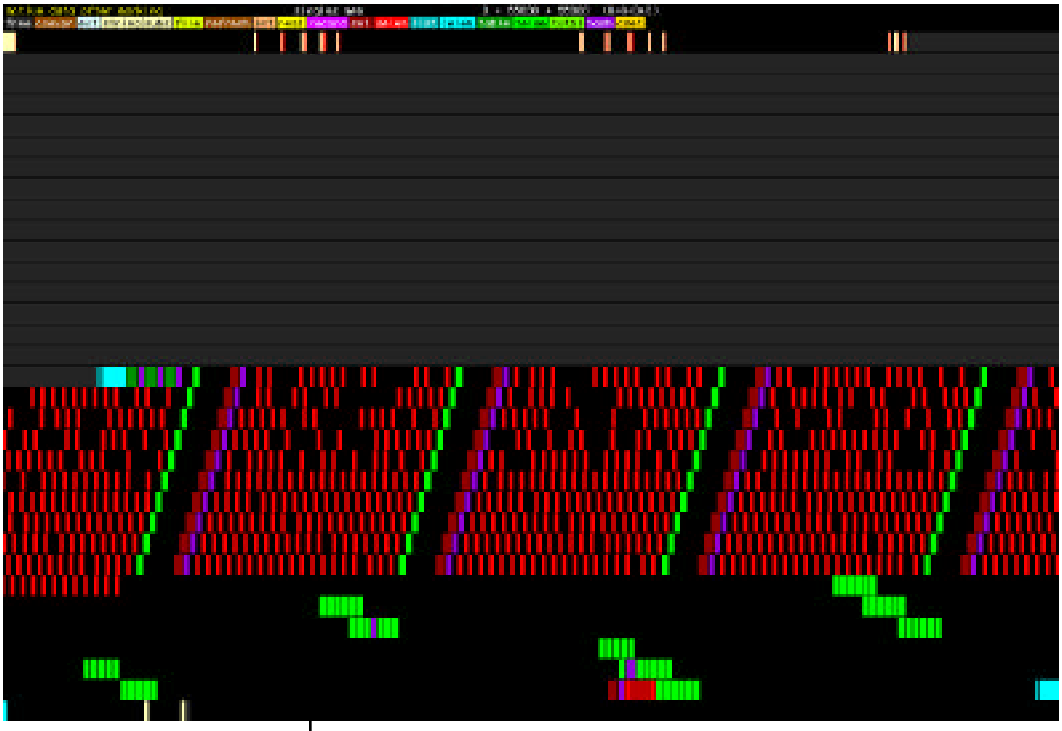
free	unused memory
coexpr	co-expression block
ext	external block
string	string
subs	substring trapped variable
file	file block
refresh	co-expression refresh block
int	large integer
real	real number
record	record
set	set header
selem	set element
list	list header
lelem	list element
table	table header
telem	table element
tvtbl	table element trapped variable
hash	hash header block
cset	cset

In color displays, each type has a different color. Here there are just different shades of gray and patterns. Although the types can hardly be distinguished in black-and-white prints, they are easily identified in color displays. You'll have to use your imagination in the description and displays that follow.

Icon's allocated storage regions follow the legend. The static region, which usually isn't very interesting, is not shown. The string region is followed by the block region. These regions are shown as being contiguous. They are contiguous in some implementations but not in others. Consequently, you shouldn't interpret the contiguity of these two regions as being significant.

The allocated part of the string region shows strings in white with gray bars marking their ends. The unallocated portion of the string region is darker gray.

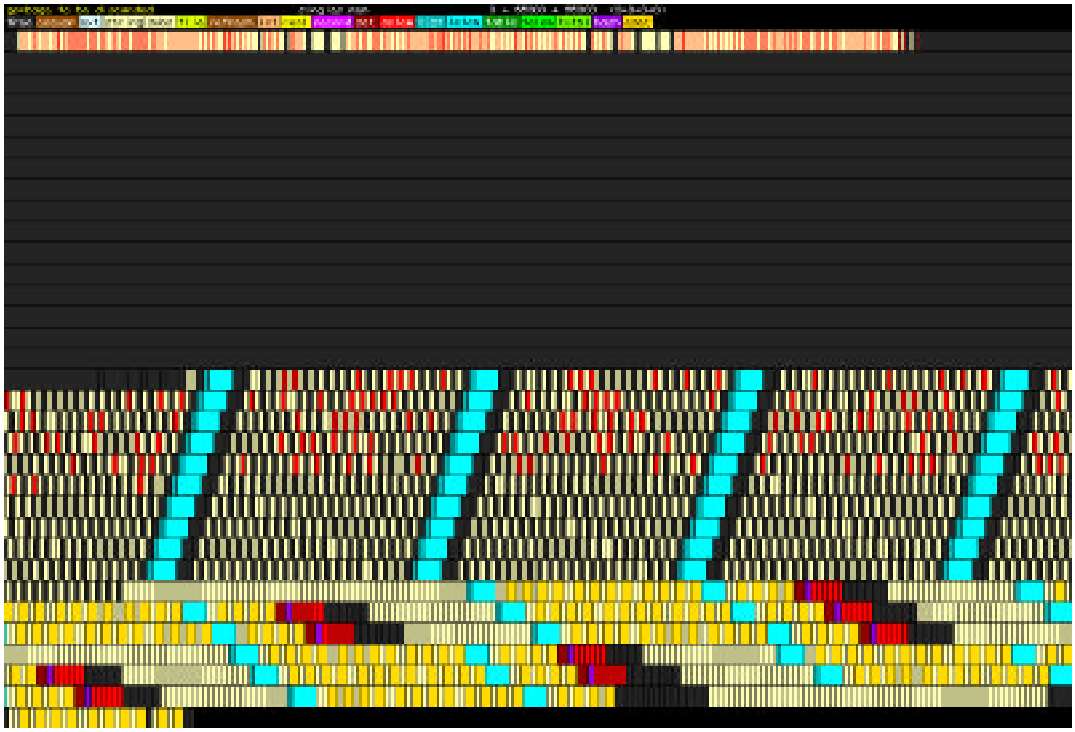
The block region in the display above is fully allocated and a garbage collection is about to take place. This is one of the significant times in storage management at which snapshots are taken. As described previously, the first phase of garbage collection “marks” all space that needs to be saved. The completion of marking is another significant point in storage management and is shown below:



The strings and blocks that need to be saved are shown in black. Notice that most of the strings that need to be saved are at the beginning of the string region. The unmarked strings represent transient allocation and storage throughput in the string region. Similarly, most of the blocks in the block region are near the beginning (the program that produced this allocation history file maintains a large table). The rest of the blocks, except for a couple near the end that were recently allocated and

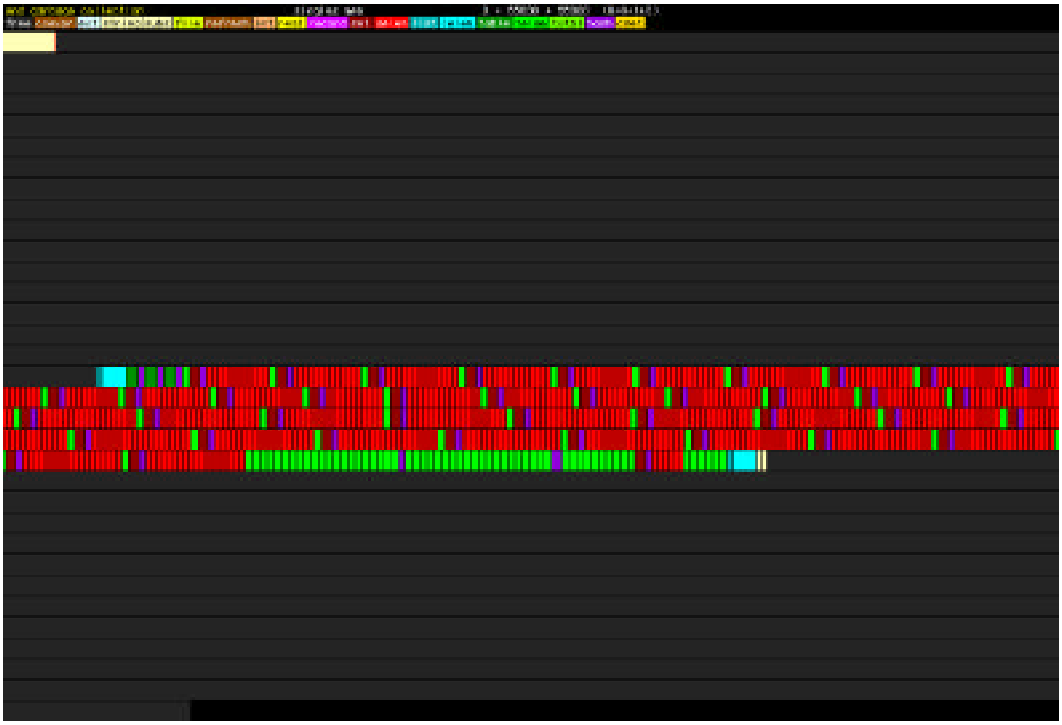
still in use, represent storage throughput in the block region.

The display below shows the same storage configuration with the coloring reversed — the space to be collected is now shown in black. (Notice the optical illusion that makes some blocks appear to be crooked.)



Finally, the display below shows memory after the saved space has been compacted to the beginnings of the regions. The space in the string region is shown as a single string. Since overlapping substrings may be created after strings are allocated, the identification of the ends of strings after garbage collection is somewhat problematical and is not shown.

Notice that in the legend, a garbage collection is now attributed to the need for space in the block region. The string region was collected too, but not “charged” for it. Observe that garbage collection has freed a considerable amount of space for subsequent allocation.



Running MemMon

By now, you should be ready to run MemMon so that you can see displays like these in real life. These displays will be much more interesting than the pictures here, especially if you have a color monitor.

There are two ways to start MemMon — opening it as an application or opening an allocation history file, which launches MemMon automatically. Start by opening the concord.mem file in the MemMon Samples folder from the ProIcon distribution disk.

You'll see a display that is similar to the ones given here, but instead of just a snapshot, you'll see each object as it is allocated. The display pauses when a garbage collection is needed. This gives you a chance to look at the whole display in detail.

Look at the individual objects that have been allocated and see if you can identify their types from the legend. Click on an object and the corresponding legend box blinks twice. This feature is especially useful if you are using a black-and-white monitor.

When you're ready to watch the garbage collection, press the space bar. The block region is marked first, followed by the string region. You'll see each object turn black as it's marked — indicating that it is to be saved. The display pauses again after all the objects to be saved have been marked. The unmarked objects are "garbage" and they will be collected. When you press the space bar again, you'll see the same display, but with marked objects in their original colors and the unmarked objects in black. Press the space bar once again and you'll see the results of compacting the saved objects toward the beginning of the storage regions. Press the space bar one more time and you'll see allocation begin again as the program continues after the garbage collection.

That's the basic idea in running MemMon. But there are several more things you can do.

Snapshots

Anytime there is a MemMon display on the screen, even when the display is running, you can take a snapshot of the display. The result is a color PICT file that you can view separately, import into a document, and so forth.

The keyboard sequence ⌘-T takes a snapshot. When you take the first snapshot, a dialog box is presented to allow you to specify the location and name of the file to hold the snapshot. The normal file-name suffix is .001 for the first snapshot. You are not prompted for names of subsequent snapshots; the suffix on your original choice is incremented automatically to produce .002, .003, and so forth.

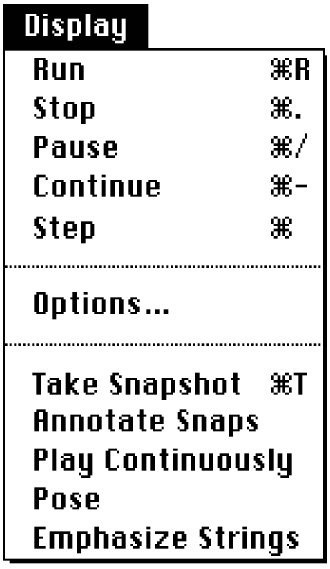
You can also arrange to have snapshots taken automatically at specified points in a display. See the Options specifications in the description of the Display menu that follows.

Menus

The File menu lets you close the current allocation history file, open another one, and quit. Some of the options in this menu are grayed out and unavailable. For example, you can't print from MemMon. There also are options related to palettes that are reserved for future use:

All the items in the Edit menu are grayed out; you can't do things like cutting and pasting in MemMon.

The Display menu lets you control the display and set options:



The top five menu items refer to running the display. If you launch MemMon by opening an allocation history file, the display starts running automatically. If you launch the MemMon application directly and then open an allocation history file, you'll need to select Run (or use the keyboard shortcut ⌘-R) to start the display. You also can use Run to start the display over after it's finished.

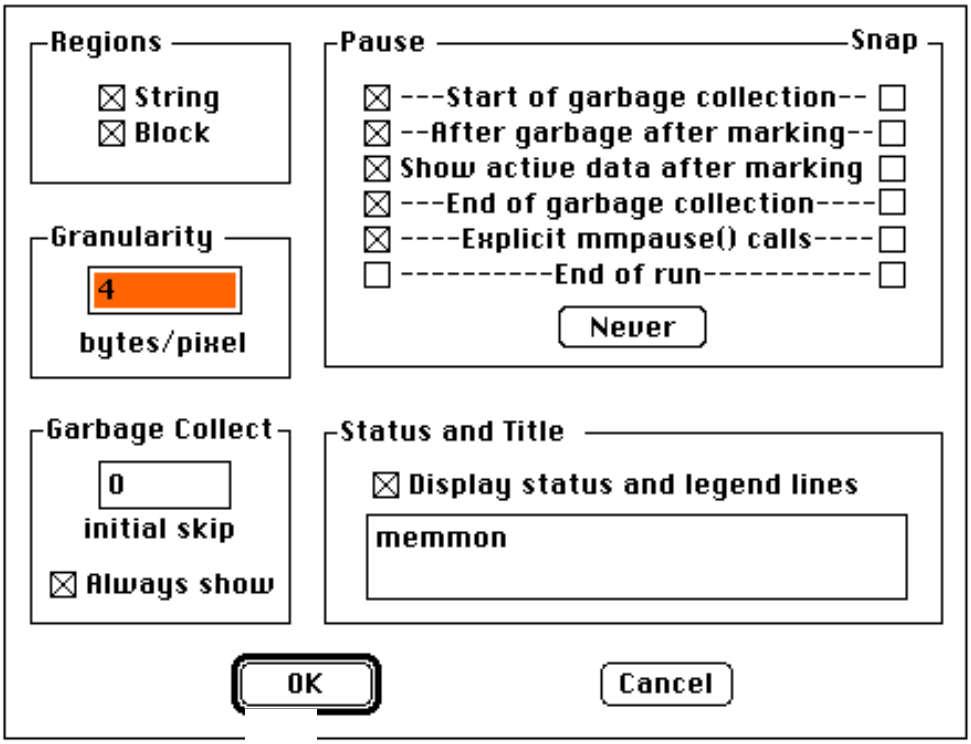
You can stop the display at any time by selecting Stop from the Display menu. It's faster and usually more convenient to use the keyboard shortcut ⌘-. .

If you want to stop the display temporarily, select Pause or use ⌘-/ . To go on, select Continue or use ⌘-– .

If you want to see each step in the allocation or marking process, select Step or use ⌘- (the ⌘ key in combination with the space bar).

The Options menu lets you control the display and snapshots. *Important note:* options must be specified before a display starts to run; changing them in mid-

stream has no effect on the current display.



The default values are as shown above. You can change any of them. The Regions box specifies which allocated data regions are displayed. Normally both the string and block regions are displayed, but you can chose to have only one displayed by unchecking the box for the other. If only one region is selected, the height of the display lines is scaled up to fit the window.

The Pause / Snap box controls when the display pauses and when automatic screen snapshots are taken. The first four selections refer to the states of garbage collection. The fifth selection refers to the Icon function mmpause(), which can be used to cause the display to pause at times other than the normal ones. Click on Never to disable all pauses and snapshots.

The Granularity box controls the resolution of the display with respect to memory space. Normally four

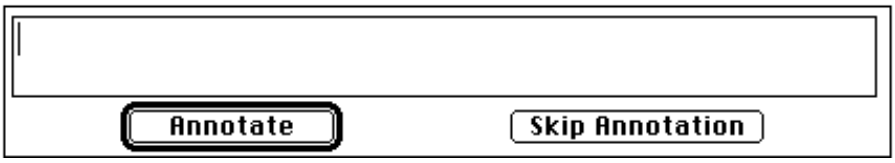
bytes of memory are shown as one pixel horizontally on the display. You can change this value to get a higher or lower resolution. The height of the display lines is scaled to fit the display window.

The Garbage Collect box lets you specify when the display begins with respect to the number of garbage collections. The default is 0, so the display normally starts at the beginning of program execution. If you specify a larger number, the display does not start until after that many garbage collections. If Always Show is checked, the marking phase of garbage collection is always displayed. If this box is not checked, the marking phase is bypassed if none of the Pause/Snap boxes related to marking is checked.

The Status and Title box determines whether or not the two legend lines are shown. They can be omitted from displays by unchecking that box. The title, which appears in the middle of the first legend line, normally is the name of the allocation history file to be displayed. The title can be changed by editing the text box.

Selecting Take Snapshot is equivalent to ⌘-T and produces a PICT snapshot of the current display.

Checking Annotate Snaps causes a text box to be



displayed before each snapshot is taken, allowing you to provide about 100 characters of identifying information:

After you have entered the annotation, click Annotate and the annotated snapshot will be taken.

If you want to annotate some pictures in a run but not others, don't change the status of Annotate Snaps from picture to picture. Instead, leave it checked and click on Skip Annotation when the annotation box is presented for a picture you do not wish to annotate.

Play Continuously causes the display to start over after the end of the allocation history file is reached.

Pose changes the screen background to black so that the display can be seen without the usual underlying desktop in the background.

Emphasize Strings changes the method by which strings in the allocated string region are displayed to make them more obvious.

F

Further Reading

Further Reading

Bond, Gary. *XCMD's for HyperCard* Portland, Oregon: MIS: Press, 1988. ISBN 0-943518-85-7.

Corré, Alan D. *Icon Programming for Humanists* Englewood Cliffs, New Jersey: Prentice-Hall, 1990. ISBN 0-13-450180-2.

Griswold, Madge T. and Griswold, Ralph E., eds. *The Icon Analyst*. Icon Project, Department of Computer Science, Gould-Simpson Building, The University of Arizona, Tucson, Arizona 85721.

Griswold, Madge T. and Griswold, Ralph E., eds. *The Icon Newsletter*. Icon Project, Department of Computer Science, Gould-Simpson Building, The University of Arizona, Tucson, Arizona 85721.

Griswold, Ralph E. and Griswold, Madge T. *The Icon Programming Language*. Second edition. Englewood Cliffs, New Jersey: Prentice-Hall, 1990. ISBN 0-13-447889-4.

Griswold, Ralph E. and Griswold, Madge T. *The Implementation of the Icon Programming Language* Princeton, New Jersey: Princeton University Press, 1986. ISBN 0-691-08431-9.

Griswold, R. E.; Poage, J. F.; and Polonsky, I. P. *The SNOBOL4 Programming Language, second edition.* Englewood Cliffs, New Jersey: Prentice-Hall 1971. ISBN 13-815373-6.

Hockey, Susan. *Snobol Programming for the Humanities* Oxford, England: Clarendon Press. 1985. ISBN 0-19-824675-7.

Macintosh. *HyperCard Script Language Guide; The HyperTalk language*. Reading, Massachusetts: Addison-Wesley Publishing Company, Inc. 1988. ISBN-0-201-17632-7.

Index

Index

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Z