Version 8.10 of the Icon Programming Language

Ralph E. Griswold, Clinton L. Jeffery, and Gregg M. Townsend

Department of Computer Science, The University of Arizona

1. Introduction

The current version of Icon is Version 8.10. The second edition of the Icon book [1] describes Version 8.0. This description is a supplement to that book.

Most of the language extensions in Version 8.10 are upward-compatible with previous versions of Icon and most programs written for earlier versions work properly under Version 8.10. The language additions to Version 8.10 are:

- a preprocessor
- an optional interface to graphic facilities (for platforms that support them)
- new functions and keywords
- several minor changes

There also are changes to the implementation in Version 8.10, including support for multiple storage regions, that provide more capabilities for some users. See Section 3.

2. Language Features

2.1 Preprocessing

All Icon source code passes through a preprocessor before interpretation. The effects of preprocessing can be seen by running `icont` or `iconc` with the `–E` flag. Preprocessing by *m4*, if selected, precedes Icon's preprocessing and does not apply to included files.

Preprocessor lines control the actions of the preprocessor and are not passed to the Icon translator or compiler. If no preprocessor lines are present, the source code passes through the preprocessor unaltered.

A source line is a preprocessor line if its first non-whitespace character is a `$` and if that `$` is not followed by another punctuation character. The general form of a preprocessor line is

`$` *directive arguments* `# comment`

Whitespace separates tokens when needed, and case is significant, as in Icon proper. The entire preprocessor directive must appear on a single line which cannot be continued. The comment portion is optional. An invalid preprocessor line produces an error except when skipped by conditional compilation.

Preprocessor lines can appear anywhere in an Icon source file without regard to procedure, declaration, or expression boundaries.

Include Directives

An include directive has the form

`$include` *filename*

An include directive causes the contents of another file to be interpolated in the source file. The file name must be quoted if it is not in the form of an Icon identifier. `#line` comments are inserted before and after the included file to allow proper identification of errors.

Included files may be nested to arbitrary depth, but a file may not include itself either directly or indirectly. File names are looked for first in the current directory and then in the directories listed in the environment variable `LPATH`. Relative paths are interpreted in the preprocessor's context and not in relation to the including file's location.

Line Directives

A line directive has the form

`$line` *n* [*filename*]

The line containing the preprocessing directive is considered to be line *n* of the given file (or the current file, if unspecified) for diagnostic and other purposes. The line number is a simple unsigned integer. The file name must be quoted if it is not in the form of an Icon identifier.

Note that the interpretation of *n* differs from that of the C preprocessor, which interprets it as the number of the *next* line.

`$line` is an alternative form of the older, special comment form `#line`. The preprocessor recognizes both forms and produces the fully specified older form for the lexical analyzer.

Define Directives

A define directive has the form

`$define` *name text*

The define directive defines the text to be substituted for later occurrences of the identifier *name* in the source code. *text* is any sequence of characters except that any string or cset literals must be properly terminated within the definition. Leading and trailing whitespace are not part of the definition. The text can be empty.

Redefinition of a name is allowed only if the new text is exactly the same as the old text. For example, `3.0` is not the same as `3.000`.

Definitions remain in effect through the end of the current original source file, crossing include boundaries, but they do not persist from one command-line file to another.

Note that the Icon preprocessor, unlike the C preprocessor, does not provide parameterized definitions. If the text begins with a left parenthesis, it must be separated from the name by at least one space.

Undefine Directives

An undefine directive has the form

`$undef` *name*

The current definition of *name* is removed, allowing its redefinition if desired. It is not an error to undefine a nonexistent name.

Predefined Symbols

At the start of each source file, several symbols are automatically defined to indicate the Icon system configuration. Each potential predefined symbol corresponds to one of the values produced by the keyword `&features`. If a feature is present, the symbol is defined with a value of 1. If a feature is absent, the symbol is not defined.

predefined symbol	&features value
_AMIGA	Amiga
_ACORN	Acorn Archimedes
_ATARI	Atari ST
_CMS	CMS
_MACINTOSH	Macintosh
_MSDOS_386	MS–DOS/386
_MSDOS	MS–DOS
_MVS	MVS
_OS2	OS/2
_PORT	PORT
_UNIX	UNIX
_VMS	VMS
_COMPILED	compiled
_INTERPRETED	interpreted
_ASCII	ASCII
_EBCDIC	EBCDIC
_EXPANDABLE_REGIONS	expandable regions
_FIXED_REGIONS	fixed regions
_CALLING	calling to Icon
_CO_EXPRESSIONS	co–expressions
_DIRECT_EXECUTION	direct execution
_EVENT_MONITOR	event monitoring
_EXECUTABLE_IMAGES	executable images
_EXTERNAL_FUNCTIONS	external functions
_KEYBOARD_FUNCTIONS	keyboard functions
_LARGE_INTEGERS	large integers
_MEMORY_MONITOR	memory monitoring
_MULTITASKING	multiple programs
_MULTIREGION	multiple regions
_PIPES	pipes
_RECORD_IO	record I/O
_STRING_INVOKE	string invocation
_SYSTEM_FUNCTION	system function
_VISUALIZATION	visualization support
_WINDOW_FUNCTIONS	window functions
_X_WINDOW_SYSTEM	X Windows
_PRESENTATION_MGR	Presentation Manager
_ARM_FUNCTIONS	Archimedes extensions
_DOS_FUNCTIONS	MS–DOS extensions

Predefined symbols have no special status: like other symbols, they can be undefined and redefined.

A few of the entries in &features that are always available, such as error traceback, lack corresponding predefined symbols.

Substitution

As input is read, each identifier is checked to see if it matches a previous definition. If it does, the value replaces the identifier in the input stream.

No whitespace is added or deleted when a definition is inserted. The replacement text is scanned for defined identifiers, possibly causing further substitution, but recognition of the original identifier name is disabled to prevent infinite recursion.

Occurrences of defined names within comments, literals, or preprocessor lines are not altered.

Conditional Compilation

Conditional compilation directives have the form

```
$ifdef name
```

and

```
$ifndef name
```

`$ifdef` or `$ifndef` cause subsequent code to be accepted or skipped depending on whether *name* has been previously defined. `$ifdef` succeeds if a definition exists; `$ifndef` succeeds if a definition does *not* exist. The value of the definition does not matter.

A conditional block has this general form:

```
$ifdef name    or    $ifndef name
   ... code to use if test succeeds ...
$else
   ... code to use if test fails ...
$endif
```

The `$else` section is optional. Conditional blocks can be nested provided that all of the `$if`/`$else`/`$endif` directives for a particular block are in the same source file. This does not prevent the conditional inclusion of other files via `$include` as long as any included conditional blocks are similarly self-contained.

Error Directives

An error directive has the form

```
$error text
```

An `$error` directive forces a fatal compilation error displaying the given text. This is typically used with conditional compilation to indicate an improper set of definitions.

Subtle Points

Because substitution occurs on replacement text but not on preprocessor lines, either of the following sequences is valid:

```
$define x 1           $define y x
$define y x           $define x 1
write(y)              write(y)
```

It is possible to construct pathological examples of definitions that combine with the input text to form a single Icon token, as in

```
$define X e3          $define Y 456e
write(123X)           write(Y+3)
```

If the value of a definition is an expression, it is wise to parenthesize it so that precedence causes no problems when it is substituted.

Redefinition of Icon's reserved words is allowed but not advised. Redefinition of an identifier associated with a keyword is similarly dangerous.

The preprocessor is ignorant of multi-line literals and can potentially be fooled this way into making a substitution inside a string constant.

The preprocessor works hard to get line numbers right, but column numbers are likely to be rendered incorrect by substitutions.

Substitution cannot produce a preprocessor directive. By then it is too late.

2.2 Graphic Facilities

Version 8.10 provides support for graphic facilities through a combination of high-level support and a repertoire of functions. The name X-Icon is used to distinguish versions of Icon that support these facilities. *Note:* There are several changes to X-Icon in Version 8.10. Persons who used an earlier version of X-Icon should consult the current reference manual [2].

2.3 New Functions and Keywords

The new functions and keywords are described briefly here. At the end of this report there also is a sheet with more complete descriptions in the style of the second edition of the Icon book. This sheet can be trimmed and used as an insert to the book.

There are five new functions:

`chdir(s)`	Changes the current directory to `s` but fails if there is no such directory or if the change cannot be made.
`delay(i)`	Delays execution `i` milliseconds. Delaying execution is not supported on all platforms; if it is not, there is no delay and `delay()` fails.
`flush(f)`	Flushes the output buffers for file `f`.
`function()`	Generates the names of the Icon (built-in) functions.
`sortf(X,i)`	Produces a sorted list of the elements of `X`. The results are similar to those of `sort(X,i)`, except that among lists and among records, structure values are ordered by comparing their `i`th fields.

There are five new keywords:

`&allocated`	Generates the number of bytes allocated since the beginning of program execution. The first result is the total number of bytes in all regions, followed by the number of bytes in the static, string, and block regions.
`&e`	The base of the natural logarithms, 2.71828 ...
`&phi`	The golden ratio, 1.61803 ...
`&pi`	The ratio of the circumference of a circle to its diameter, 3.14159 ...
`&progname`	The file name of the executing program. `&progname` is a variable and a string value can be assigned to it to replace its initial value.

The X interface adds additional new keywords [2].

Some UNIX platforms now support the keyboard functions `getch()`, `getche()`, and `kbhit()`. Whether or not these functions are supported can be determined from the values generated by `&features`. *Note:* On UNIX platforms, ‘‘keyboard’’ input comes from standard input, which may not necessarily be the keyboard. *Warning:* The keyboard functions under UNIX may not work reliably in all situations and may leave the console in a strange mode if interrupted at an unfortunate time. These potential problems should be kept in mind when using these functions.

2.4 Other Enhancements

Record Subscripting

A record can now be subscripted by the string name of one of its fields, as in

```
z["r"]
```

which is equivalent to

```
z.r
```

If the named field does not exist for the record, the subscripting expression fails.

Multiple Subscripts

Multiple subscripts are now allowed in subscripting expressions. For example,

```
L[i, j, k]
```

is equivalent to

```
L[i][j][k]
```

Named Functions

The function `proc(x, i)` has been extended so that `proc(x, 0)` produces the built-in function named `x` even if the global identifier having that name has been assigned another value. `proc(x, 0)` fails if `x` is not the name of a function.

2.5 Minor Changes

- The `dynamic` declaration, a synonym for `local`, is no longer supported.
- The `invocable` declaration is accepted but ignored by the interpreter to provide source-language compatibility with the Icon compiler. See [3] for a description of this declaration.
- Real literals that are less than 1 no longer need a leading zero. For example, `.5` now is a valid real literal instead of being the dereferencing operator applied to the integer 5.
- The identifiers listed by `display()` are now given in sorted order.
- In sorting structures, records now are first sorted by record name and then by age (serial number).
- The keyword `&features` now includes either `interpreted` or `compiled` to indicate whether the program is interpreted or compiled.
- If X-Window facilities are supported, `&features` also includes `X Windows`.
- If multiple storage regions are supported, `&features` also includes `Multiple Regions`.
- Error message 101 now reads `integer expected or out of range` to reflect the fact that not all operations support large integers.
- Error 120 now reads `two csets or two sets expected` to more accurately reflect the fact that set operations require arguments of the same type.
- Error 125, `list or set expected`, has been added for `sortf()`.
- Errors 140, `window expected`, and 141, `program terminated by window manager`, have been added when X-Window facilities are supported.
- Errors 316, `interpreter stack too large`, and 318, `co-expression stack too large`, have been added for 16-bit platforms.

3. Implementation Changes

The implementation of Version 8.10 is different in many respects from the implementation of Version 8.0. The most significant differences are:

- Icon now uses fixed-sized storage regions. Multiple regions are allocated if needed.
- Memory monitoring is no longer supported.
- Under MS-DOS, `iconx` now finds icode files at any place on the `PATH` specification as well as in the current directory.

In addition, the tables used by `icont` now expand automatically. The `–S` option is no longer needed. As a side effect of this change, the sizes of procedures are no longer listed during translation.

4. Limitations, Bugs, and Problems

- 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

&allocated : i1, i2, i3, i4 cumulative allocation

&allocated generates the total amount of space, in bytes, allocated since the beginning of program execution. The first value is the total for all regions, followed by the totals for the static, string, and block regions, respectively. The space allocated in the static region is always given as zero. *Note:* &allocated gives the cumulative allocation; &storage gives the current allocation; that is, the amount that has not been freed by garbage collection.

&e : r base of natural logarithms

The value of &e is the base of the natural logarithms, 2.71828 ... .

&phi : r golden ratio

The value of &phi is the golden ratio, 1.61803 ... .

&pi : r ratio of circumference to diameter of a circle

The value of &pi is the ratio of the circumference of a circle to its diameter, 3.14159 ... .