ICI Technical Description

Version 1.2

Tim Long

The basic execution model

The ICI interpreter's *execution engine* calls on the *parser* to read and compile a statement from an input stream. The parser in turns calls on the *lexical analyser* to read tokens. Upon return from the parser the execution engine executes the compiled statement. When the statement has finished execution, the execution engine repeats the sequence.

The lexical analyser

The ICI lexical analyser breaks the input stream into tokens, optionally separated by white-space (which includes comments as described below). The next token is always the longest string of following characters which could possibly be a token. The following are tokens:

```
/       /=      $       @       (       )       {       }
,       ~       ~~      ~~=     ~~~     [       ]       .
*       *=      %       %=      ^       ^=      +       +=
++      -       -=      --      ->      >       >=      >>
>>=     <       <=      <=>     <<      <<=     =       ==
!       !=      !~      &       &&      &=      |       ||
|=      ;       ?       :
```

The following are also tokens:

- The character '#' followed by any sequence of characters except a newline, then another '#'. This token is a *regular-expression*.
- The character ' (single quote) followed by a single character (other than a newline) or a single *backslash character sequence* (described below), followed by another single quote. This token is a *character-code*. A single quote followed by other than the above sequence will result in an error.
- The character " (double quote) followed by any sequence of characters (other than a newline) and *backslash character sequences*, up to another double quote character. This token is a *string*.

A *backslash character sequence* is any of the following:

`\n`	newline (ASCII 0x0A)
`\t`	tab (ASCII 0x09)
`\v`	vertical tab (ASCII 0x0B)
`\b`	back space (ASCII 0x08)
`\r`	carriage return (ASCII 0x0D)
`\f`	form feed (ASCII 0x0C)
`\a`	audible bell (ASCII 0x07)
`\e`	escape (ASCII 0x1B)
`\\`	backslash (ASCII 0x5C)
`\′`	single quote (ASCII 0x27)
`\″`	double quote (ASCII 0x22)
`\?`	question mark (ASCII 0x3F)
`\c`*x*	control-*x*
`\x`*x*`..`	the character with hex code *x*...
`\`*n*	the character with octal code *n*. (1, 2 or 3 octal digits)

Consecutive string-literals, seperated only by white-space, are concatenated to form a single strings-literal.

- Any upper or lower case letter, any digit, or '_' (underscore) followed by any number of the same (or other characters which may be involved in a floating point number while that is a valid interpretation). A token of this form may be one of three things:

  If it can be interpreted as an integer, it is an *integer-number*.

  Otherwise, if it can be interpreted as a floating point number, it is a *floating-point-number*.

  Otherwise, it is an *identifier*.

Notice that keywords are not recognised directly by the lexical analyser. Instead, certain identifiers are recognised as keywords by the parser as described below.

Comments (which are white-space) are started with the characters /* and continue until the next */. Also, lines which start with a # character are ignored.

An introduction to variables, modules and scope

Variables are simple identifiers which have a value associated with them. They are in themselves typeless, depending on the type of the value currently assigned to them.

The term *module* in ICI refers to a collection of functions, declarations and code which share the same variables. Typically each source file is a module, but not necessarily.

In ICI, modules may be nested in a hierarchical fashion. Within a module, variables can be declared as either *static* or *extern*. When a variable is declared as static it is visible to code defined in the module of its definition, and to code defined in sub-modules of that one. This is termed the *scope* of the variable.

When a variable is defined as *extern* it is declared *static* in the parent module. Thus the parent module and all sub-modules of the parent module have that variable in their scope. Variables of this type, whether originally declared extern or static, will be henceforward referred to as static variables.

Static variables are persistent variables. That is they remain in existence even when execution completely leaves their scope, despite not being visible to any executing code. They are visible again when code flow again enters their scope.

The scoping of static variables is strictly governed by the nesting of the modules, not by the flow of execution. For example. Suppose two neighbouring modules (call them module **A** and module **B**) each define a variable called **theVariable**. When some code in module **A** calls a function defined in module **B** and that function refers to **theVariable**; it is referring to the version of **theVariable** defined in module **B**, not the one defined in module **A**.

Variables in sub scopes hide variables of the same name defined in outer scopes.

The second type of variable in ICI is the *automatic*, or *auto*, variable. Automatic variables are not persistent. They last only as long as a module is being parsed or a function is being executed. For instance, each time a function is entered a copy is made of the auto variables which were declared in the function. This group of variables generally only persists during the execution of the function; once the function returns they are discarded.

The parser

The parser uses the lexical analyser to read a source input stream. The parser also has reference to the variable-scope within which this source is being parsed, so that it may define variables.

The parser will define variables within the current scope, and, when code is parsed at the outermost level, return it to the execution engine for execution.

For some constructs the parser will in turn call upon the execution engine to evaluate a sub-construct within a statement.

The following sections will work through the syntax of ICI with explanations and examples. Occasionally constructs will be used ahead of their full explanation. Their intent should be obvious.

The following notation is used in the syntax in these sections. Note that the syntax given in the text is not always exact, but rather designed to aid comprehension. The exact syntax is given in a later section.

bold	The **bold** text is literal ASCII text.
italic	The *italic* text is a construct further described elsewhere.
[xxx *]*	The xxx is optionally present.
xxx...	The xxx may be present zero or more times.

As noted previously there are no reserved words recoginsed by the lexical anaylyser, but certain identifiers will be recognised by the parser in certain syntactic positions (as seen below). While these identifiers are not otherwise restricted, special action may need to be taken if they are used as simple variable names. They probably should be avoided. The complete list is:

NULL	auto	break	case
continue	default	do	else
extern	for	forall	if
in	onerror	return	static
switch	try	while	

We now turn our attention to the syntax itself.

Firstly consider the basic statement which is the unit of operation of the parser. As stated earlier the execution engine will call on the parser to parse one top-level statement at a time. We split the syntax of a statement into two categories (purely for semantic clarity):

statement	*executable-statement*
	declaration

That is, a statement is either an *executable-statement* or a *declaration*. We will first consider the *executable-statement*.

These are statements that, at the top-level of parsing, can be translated into code which can be returned to the execution engine. This is by far the largest category of statements:

executable-statement	*expression* ***;***
	compound-statement
	if (*expression* ***)*** *statement*
	if (*expression* ***)*** *statement* ***else*** *statement*
	while (*expression* ***)*** *statement*
	do *statement* ***while (*** *expression* ***) ;***
	for (*[expression]****;*** *[expression]****;*** *[expression]* ***)*** *statement*
	forall (*expression [, expression]* ***in*** *expression* ***)*** *statement*
	switch (*expression* ***)*** *compound-statement*
	case *parser-evaluated-expression* ***:***
	`default ;`
	`break ;`
	`continue ;`
	`return` *[expression]* **`;`**
	`try` *statement* **`onerror`** *statement*
	`;`

These are the basic executable statement types. Many of these involve *expression*s, so before examining each statement in turn we will examine the *expression*. We will do this by starting with the most primitive elements of expressions and working back up to the top level.

The lowest level building block of an expressions is the *factor*:

factor	*integer-number*
	character-code
	floating-point-number

string
regular-expression
identifier
NULL
(*expression* **)**
[array *expression-list* **]**
[set *expression-list* **]**
[struct [: *expression* **,]** *assignment-list* **]**
[func *function-body* **]**

The constructs *integer-number*, *character-code*, *floating-point-number, string*, and *regular-expression* are primitive lexical elements (described above). Each is converted to its internal form and is an object of type *int*, *int*, *float*, *string*, or *regexp* respectively.

A *factor* which is an *identifier* is a variable reference. But its exact meaning depends upon its context within the whole expression. Variables in expressions can either be placed so that their value is being looked up, such as in:

```
a + 1
```

Or they can be placed so that their value is being set, such as in:

```
a = 1
```

Or they can be placed so that their value is being both looked up and set, as in:

```
a += 1
```

Only certain types of expression elements can have their value set. A variable is the simplest example of these. Any expression element which can have its value set is termed an *lvalue* because it can appear on the left hand side of an assignment (which is the simplest expression construct which requires an lvalue). Consider the following two expressions:

```
1 = 2                   /* WRONG */
a = 2                   /* OK */
```

The first is illegal because an integer is not an lvalue, the second is legal because a variable reference is an lvalue. Certain expression elements, such as assignment, require an operand to be an lvalue. The parser checks this.

The next factor in the list above is **NULL**. The keyword NULL stands for the value NULL which is the general undefined value. It has its own type, NULL. Variables which have no explicit initialisation have an initial value of NULL. Its other uses will become obvious later in this document.

Next is the construct **(** *expression* **)**. The brackets serve merely to make the expression within the bracket act as a simple factor and are used for grouping, as in ordinary mathematics.

Finally we have the four constructs surrounded by square brackets. These are textual descriptions of more complex data items; typically known as *literals*. For example the factor:

```
[array 5, 6, 7]
```

is an array of three items, that is, the integers 5, 6 and 7. Each of these square bracketed constructs is a textual description of a data type named by the first identifier after the starting square bracket. A full explanation of these first requires an explanation of the fundamental aggregate types.

An introduction to arrays, sets and structs

There are three fundamental aggregate types in ICI: arrays, sets, and structs. Certain properties are shared by all of these (and other types as will be seen later). The most basic property is that they are each collections of other values. The next is that they may be "indexed" to reference values within them. For example, consider the code fragment:

```
a = [array 5, 6, 7];
i = a[0];
```

The first line assigns the variable a an array of three elements. The second line assigns the variable i the value currently stored at the *first* element of the array. The suffixing of an expression element by an expression in square brackets is the operation of "indexing", or referring to a sub-element of an aggregate, and will be explained in more detail below.

Notice that the *first* element of the array has index *zero*. This is a fundamental property of ICI arrays.

The next ICI aggregate we will examine is the set. Sets are unordered collections of values. Elements "in" the set are used as indexes when working with the set, and the values looked up and assigned are interpreted as a booleans. Consider the following code fragment:

```
s = [set 200, 300, "a string"];
if (s[200])
    printf("200 is in the set\n");
if (s[400])
    printf("400 is in the set\n");
if (s["a string"])
    printf("\"a string\" is in the set\n");
s[200] = 0;
if (s[200])
    printf("200 is in the set\n");
```

When run, this will print:

```
200 is in the set
"a string" is in the set
```

Notice that there was no second printing of "200 is in the set" because it was removed from the set on the third last line by assigning zero to it.

Now consider structs. Structs are unordered collections of values indexed by any values. Other properties of structs will be discussed later. The typical indexes of structs are strings. For this reason notational shortcuts exist for indexing structures by simple strings. Also, because each element of a struct is actually an index and value pair, the syntax of a struct literal is slightly different from the arrays and sets seen above. Consider the following code fragment:

```
s = [struct a = 123, b = 456, xxx = "a string"];
printf("s[\"a\"] = %d\n", s["a"]);
printf("s.a = %d\n", s.a);
printf("s.xxx = \"%s\"\n", s.xxx);
```

Will print:

```
s["a"] = 123
s.a = 123
s.xxx = "a string"
```

Notice that on the second line the structure was indexed by the string "a", but that the assignment in the struct literal did not have quotes around the *a*. This is part of the notational shortcut which will be discussed further, below. Also notice the use of *s.a* in place of s["a"]. This is a similar shortcut, also discussed below.

Back to expression syntax

The aggregate literals, which in summary are:

[`array` *expression-list* **]**
[`set` *expression-list* **]**
[`struct` *[* **:** *expression* **,** *] assignment-list* **]**
[`func` *function-body* **]**

involve three further constructs, the *expression-list,* which is a comma separated list of expressions; the *assignment-list,* which is a comma separated list of assignments; and the *function-body,* which is the argument list and code body of a function. The syntax of the first of these is:

expression-list	*empty*
	expression [**,** *]*
	expression **,** *expression-list*

The *expression-list* is fairly simple. The construct *empty* is used to indicate that the whole list may be absent. Notice the optional comma after the last expression. This is designed to allow a more consistent formatting when the elements are line based, and simpler output from programmatically produced code. For example:

```
[array
    "This is the first element",
    "This is the second element",
    "This is the third element",
]
```

The assignment list has similar features:

assignment-list	*empty*
	assignment [**,** *]*
	assignment **,** *assignment-list*
assignment	*struct-key* **=** *expression*
struct-key	*identifier*
	(*expression* **)**

Each *assignment* is either an assignment to a simple identifier or an assignment to a full expression in brackets. The assignment to an identifier is merely a notational abbreviation for an assignment to a string. The following two struct literals are equivalent:

```
    [struct abc = 4]
    [struct ("abc") = 4]
```

The syntax of a *function-body* is:

function-body	**(** *identifier-list* **)** *compound-statement*
identifier-list	*empty*
	identifier [**,** *]*
	identifier **,** *identifier-list*

That is, an *identifier-list* is an optional comma separated list of *identifiers* with an optional trailing comma. Literal functions are rare in most programs; functions are normally named and defined with a special declaration form which will be seen in more detail below. The following two code fragments are equivalent; the first is the abbreviated notation:

```
    static fred(a, b){return a + b;}
```

and:

```
    static fred = [func (a, b){return a + b;}];
```

The meaning of functions will discussed in more detail below.

Aggregates in general, and literal aggregates in particular, are fully nestable:

```
    [array
        [struct a = 1, c = 2],
        [set "a", 1.2, 3],
        "a string",
    ]
```

Note that aggregate literals are entirely evaluated by the parser. That is, each expression is evaluated and reduced to a particular value, these values are then used to build an object of the required type. For example:

```
    [struct a = sin(0.5), b = cos(0.5)]
```

Causes the functions sin and cos to be called during the parsing process and the result assigned to the keys *a* and *b* in the struct being constructed. It is possible[1] to refer to variables which may be in existence while such a literal is being parsed .

This ends our consideration of the lowest level element of an expression, the *factor*.

A simple factor may be adorned with a sequence of *primary-operation*s to form a *primary-expression*. That is:

primary-expression	*factor primary-operation...*
primary-operation	**[** *expression* **]**
	. *identifier*
	. (*expression* **)**
	-> *identifier*
	-> **(** *expression* **)**
	(*expression-list* **)**

The first *primary-operation* (above) we have already seen. It is the operation of "indexing" which can be applied to aggregate types. For example, if *xxx* is an array:

```
xxx[10]
```

refers to the element of xxx at index 10. The parser does not impose any type restrictions (because typing is dynamic), although numerous type restrictions apply at execution time (for instance, arrays may only be indexed by integers, and floating point numbers are not able to be indexed at all).

The second form, **.** *identifier*, is a notational abbreviation of **[** "*identifier*" **]** , as seen previously. Similarly the third form is again just a notational variation. Thus the following are all equivalent:

```
xxx["aaa"]
xxx.aaa
xxx.("aaa")
```

And the following are also equivalent to each other:

```
xxx[1 + 2]
xxx.(1 + 2)
```

Note that factors may be suffixed by any number of *primary-operation*s. The only restriction is that the types must be right during execution. Thus:

```
xxx[123].aaa[10]
```

1.Literal aggregates are analagous to literal strings in K&R C. And likewise they have the property that modifications to the literal are persistent. Returning to the original use of the literal after it has been modified does not magically restore it to its original value.

is legal.

The two constructs

> **->** *identifier*
> **->** **(** *expression* **)**

are again notational variations. In general, constructs of the form:

> *primary-expression* -> *identifier*
> *primary-expression* -> **(** *expression* **)**

are re-written as:

> ***(*** *** *primary-expression* ***)*** **.** *identifier*
> ***(*** *** *primary-expression* ***)*** **.** ***(*** *expression* ***)***

The unary operator ***** used here is the indirection operator, its meaning is discussed later.

The last of the *primary-operation*s:

> **(** *expression-list* **)**

is the function call operation. Although, as usual, no type checking is performed by the parser; at execution time the thing it is applied to must be a function. For example:

```
my_function(1, 2, "a string")
```

and

```
xxx.array_of_funcs[10]()
```

are both function calls. Function calls will be discussed in more detail below.

This concludes the examination of a *primary-expression*.

Primary-expressions are combined with prefix and postfix unary operators to make terms:

term	*[prefix-operator...] primary-expression [postfix-operator...]*
prefix-operator	*Any of:* *** & - + ! ~ ++ -- @ $**
postfix-operator	*Any of:* **++ --**

That is, a *term* is a *primary-expression* surrounded on both sides by any number of prefix and postfix operators. Postfix operators bind more tightly than prefix operators. Both types bind right-to-left when concatenated together. That is: -!x is the same as -(!x). As in all expression compilation, no type checking is performed by the parser, because types are an execution-time consideration.

Some of these operators touch on subjects not yet explained and so will be dealt with in detail in later sections. But in summary:

Prefix operators

*	Indirection; applied to a pointer, gives target of the pointer.
&	Address of; applied to any lvalue, gives a pointer to it.
-	Negation; gives negative of any arithmetic value.
+	Positive; no real effect.
!	Logical not; applied to 0 or NULL, gives 1, else gives 0.
~	Bit-wise complement.
++	Pre-increment; increments an lvalue and gives new value.
--	Pre-decrement; decrements an lvalue and gives new value.
@	Atomic form of; gives the (unique) read-only version of any value.
$	Immediate evaluation; see below.

Postfix operators

++	Post-increment; increments an lvalue and gives old value.
--	Post-increment; decrements an lvalue and gives old value.

One of these operators, $, is only a pseudo-operator. It actually has its effect entirely at parse time. The $ operator causes its subject expression to be evaluated immediately by the parser and the result of that evaluation substituted in its place. This is used to speed later execution, to protect against later scope or variable changes, and to construct constant values which are better made with running code than literal constants. For example, an expression involving the square root of two could be written as:

```
    x = y + 1.414213562373095;
```

Or it could be written more clearly, and with less chance of error, as:

```
    x = y + sqrt(2.0);
```

But this construct will call the square root function each time the expression is evaluated. If the expression is written as:

```
    x = y + $sqrt(2.0);
```

The square root function will be called just once, by the parser, and will be equivalent to the first form.

When the parser evaluates the subject of a $ operator it recursively invokes the execution engine to perform the evaluation. As a result there is no restriction on the activity which can be performed by the subject expression. It may reference variables, call functions or even read files. But it is important to remember that it is called at parse time. Any variables referenced will be immediately interrogated for their current value. Automatic variables of any expression which is contained in a function will not be available, because the function itself has not yet been invoked; in fact it is clearly not yet even fully parsed.

The $ operator as used above increased speed and readability. Another common use is to avoid later re-definitions of a variable. For instance:

```
    ($printf)("Hello world\n");
```

Will use the *printf* function which was defined at the time the statement was parsed, even if it is latter re-defined to be some other function. It is also slightly faster, but the difference is small when only a simple variable look-up is involved. Notice the bracketing which has been used to bind the *$* to the word *printf*. Function calls are primary operations so the *$* would have otherwise referred to the whole function call as it did in the first example.

This concludes our examination of a *term* (remember that the full meaning of other prefix and postfix operators will be discussed in later sections). We will now turn to the top level of expressions where *term*s are combined with binary operators:

expression	*term* *expression infix-operator expression*
infix-operator	*Any of:*

```
@
* / %
+ -
>> <<
< > <= >=
== != ~ !~ ~~ ~~~
&
^
|
&&
||
:
?
= += -= *= /= %= >>= <<= &= ^= |= ~~= <=>
,
```

That is, an *expression* can be a simple *term*, or two *expressions* separated by an *infix-operator*. The ambiguity amongst expressions built from several binary-operator separated expressions is resolved by assigning each operator a precedence and also applying rules for order of binding amongst equal precedence levels[2]. The lines of binary operators in the syntax rules above summarise their precedence. Operators on higher lines have higher precedence than those on lower lines. Thus 1+2*3 is the same as 1+(2*3). Operators which share a line have the same precedence. All operators except those on the second last line group left-to-right. Those on the second last line (the assignment

2.The precedences and rules are identical to those of C.

operators) group right-to-left. Thus

```
a * b / c
```

is the same as:

```
(a * b) / c
```

But:

```
a = b += c
```

is the same as:

```
a = (b += c)
```

As with unary operators, the full meaning of each will be discussed in a later section. But in summary:

Binary operators

@	Form pointer
*	Multiplication, Set intersection
/	Division
%	Modulus
+	Addition, Set union
-	Subtraction, Set difference
>>	Right shift (shift to lower significance)
<<	Left shift (shift to higher significance)
<	Logical test for less than, Proper subset
>	Logical test for greater than, Proper superset
<=	Logical test for less than or equal to, Subset
>=	Logical test for greater than or equal to, Superset
==	Logical test for equality
!=	Logical test for inequality
~	Logical test for regular expression match
!~	Logical test for regular expression non-match
~~	Regular expression sub-string extraction
~~	Regular expression multiple sub-string extraction
&	Bit-wise and
^	Bit-wise exclusive or
|	Bit-wise or
&&	Logical and
||	Logical or
:	Choice separator (must be right hand subject of **?** operator)
?	Choice (right hand expression must use **:** operator)

=	Assignment
+=	Add to
-=	Subtract from
*=	Multiply by
/=	Divide by
%=	Modulus by
>>=	Right shift by
<<=	Left shift by
&=	And by
^=	Exclusive or by
|=	Or by
~~=	Replace by regular expression extraction
<=>	Swap values
,	Multiple expression separator

This concludes our consideration of *expression*s. We will now move on to each of the executable statement types in turn.

Simple expression statements

The simple expression statement:

expression **;**

Is simply an expression followed by a semicolon. The parser translates this expression to its executable form. Upon execution the expression is evaluated and the result discarded. Typically the expression will have some side-effect such as assignment, or make a function call which has a side-effect, but there is no explicit requirement that it do so. Typical expression statements are:

```
printf("Hello world.\n");
x = y + z;
++i;
```

Note that an expression statement which could have no side-effects other than producing an error may be completely discarded and have no code generated for it.

Compound statements

The compound statement has the form:

{ *statement...* **}**

That is, a compound statement is a series of any number of statements surrounded by curly braces. Apart from causing all the sub-statements within the compound statement to be treated as a syntactic unit, it has no effect. Thus:

```
printf("Line 1\n");
{
    printf("Line 2\n");
    printf("Line 3\n");
}
printf("Line 4\n");
```

When run, will produce:

```
    Line 1
    Line 2
    Line 3
    Line 4
```

Note that the parser will not return control to the execution engine until all of a top-level compound statement has been parsed. This is true in general for all other statement types.

The *if* statement

The *if* statement has two forms:

if (*expression* ***)*** *statement*
if (*expression* ***)*** *statement* ***else*** *statement*

The parser converts both to an internal form. Upon execution, the *expression* is evaluated. If the expression evaluates to anything other than 0 (integer zero) or NULL, the following statement is executed; otherwise it is not. In the first form this is all that happens, in the second form, if the expression evaluated to 0 or NULL the statement following the *else* is executed; otherwise it is not.

The interpretation of both 0 and NULL as false, and anything else as true, is common to all logical operations in ICI. There is no special boolean type.

The ambiguity introduced by multiple if statements with an lesser number of else clauses is resolved by binding else clauses with their closest possible if. Thus:

```
    if (a) if (b) dox(); else doy();
```

If equivalent to:

```
    if (a)
    {
        if (b)
            dox();
        else
            doy();
    }
```

The *while* statement

The *while* statement has the form:

while (*expression* **)** *statement*

The parser converts it to an internal form. Upon execution a loop is established. Within the loop the *expression* is evaluated, and if it is false (0 or NULL) the loop is terminated and flow of control continues after the *while* statement. But if the *expression* evaluates to true (not 0 and not NULL) the *statement* is executed and then flow of control moves back to the start of the loop where the test is performed again (although other statements, as seen below, can be used to modify this natural flow of control).

The *do-while* statement

The *do-while* statement has the following form:

do *statement* **while (** *expression* **) ;**

The parser converts it to an internal form. Upon execution a loop is established. Within the loop the *statement* is executed. Then the *expression* is evaluated and if it evaluates to true, flow of control resumes at the start of the loop. Otherwise the loop is terminated and flow of control resumes after the *do-while* statement.

The *for* statement

The *for* statement has the form:

for (*[expression]***;** *[expression]***;** *[expression]* **)** *statement*

The parser converts it to an internal form. Upon execution the first *expression* is evaluated (if present). Then, a loop is established. Within the loop: If the second *expression* is present, it is evaluated and if it is false the loop is terminated. Next the *statement* is executed. Finally, the third *expression* is evaluated (if present) and flow of control resumes at the start of the loop. For example:

```
for (i = 0; i < 4; ++i)
    printf("Line %d\n", i);
```

When run will produce:

```
Line 0
Line 1
Line 2
Line 3
```

The *forall* statement

The *forall* statement has the form:

forall (*expression [* **,***expression]* **in** *expression* ***)*** *statement*

The parser converts it to an internal form. In doing so the first and second *expression*s are required to be lvalues (that is, capable of being assigned to). Upon execution the first expression is evaluated and that storage location is noted. If the second *expression* is present the same is done for it. The third *expression* is then evaluated and the result noted; it must evaluate to an array, a set, a struct, a string, or NULL; we will call this *the aggregate*. If this is NULL, the *forall* statement is finished and flow of control continues after the statement; otherwise, a loop is established.

Within the loop, an element is selected from the noted aggregate. The value of that element is assigned to the location given by the first expression. If the second expression was present, it is assigned the key used to access that element. Then the statement is executed. Finally, flow of control resumes at the start of the loop.

Each arrival at the start of the loop will select a different element from the aggregate. If no as yet unselected elements are left, the loop terminates. The order of selection is predictable for arrays and strings, namely first to last. But for structs and sets it is unpredictable. Also, while changing the values of the structure members is acceptable, adding or deleting keys, or adding or deleting set elements during the loop will have an unpredictable effect on the progress of the loop.

As an example:

```
    forall (colour in [array "red", "green", "blue"])
        printf("%s\n", colour);
```

when run will produce:

```
    red
    green
    blue
```

And:

```
    forall (value, key in [struct a = 1, b = 2, c = 3])
        printf("%s = %d\n", key, value);
```

when run will produce (possibly in some other order):

```
    c = 3
    a = 1
    b = 2
```

Note in particular the interpretation of the value and key for a set. For consistency with the access method and the behavior of structs and arrays, the values are all 1 and the elements are regarded as the keys, thus:

```
    forall (value, key in [set "a", "b", "c"])
        printf("%s = %d\n", key, value);
```

when run will produce:

```
    c = 1
    a = 1
    b = 1
```

But as a special case, when the second expression is omitted, the first is set to each "key" in turn, that is, the elements of the set. Thus:

```
    forall (element in [set "a", "b", "c"])
```

```
        printf("%s\n", element);
```

when run will produce:

```
    c
    a
    b
```

When a forall loop is applied to a string (which is not a true aggregate), the "sub-elements" will be successive one character sub-strings.

Note that although the sequence of choice of elements from a set or struct is at first examination unpredictable, it will be the same in a second forall loop applied without the structure or set being modified in the interim.

The *switch*, *case*, and *default* statements

These statements have the forms:

> **switch (** *expression* **)** *compound-statement*
> **case** *expression* **:**
> *default* **:**

The parser converts the switch statement to an internal form. As it is parsing the compound statement, it notes any *case* and *default* statements it finds at the top level of the compound statement. When a *case* statement is parsed the *expression* is evaluated immediately by the parser. As noted previously for parser evaluated expressions, it may perform arbitrary actions, but it is important to be aware that it is resolved to a particular value just once by the parser. As the *case* and *default* statements are seen their position and the associated expressions are noted in a table.

Upon execution, the *switch* statement's *expression* is evaluated. This value is looked up in the table created by the parser. If a matching *case* statement is found, flow of control immediately moves to immediately after that *case* statement. If there is a *default* statement, flow of control immediately moves to just after that. If there is no matching *case* and no *default* statement, flow of control continues just after the entire *switch* statement.

For example:

```
    switch ("a string")
    {
    case "another string":
        printf("Not this one.\n");
    case 2:
        printf("Not this one either.\n");
    case "a string":
        printf("This one.\n");
    default:
        printf("And this one too.\n");
    }
```

When run will produce:

```
This one.
And this one too.
```

Note that the case and default statements, apart from the part they play in the construction of the look-up table, do not influence the executable code of the compound statement. Notice that once flow of control had transferred to the third case statement above, it continued through the default statement as if it had not been present. This behavior can be modified by the *break* statement described below.

It should be noted that the "match" used to look-up the switch expression against the case expressions is the same as that used for structure element look-up. That is, to match, the switch expression must evaluate to the same object as the case expression. The meaning of this will be made clear in a later section.

The *break* and *continue* statements

The *break* and *continue* statements have the form:

```
break ;
continue ;
```

The parser converts these to an internal form. Upon execution of a break statement the execution engine will cause the nearest enclosing loop (a while, do, for or forall) or switch statement within the same scope to terminate. Flow of control will resume immediately after the affected statement. Note that a *break* statement without a surrounding loop or *switch* in the same function or module is illegal.

Upon execution of a *continue* statement the execution engine will cause the nearest enclosing loop to move to the next iteration. For *while* and *do* loops this means the test. For *for* loops it means the step, then the test. For *forall* loops it means the next element of the aggregate.

The *return* statement

The *return* statement has the form:

return *[expression]* **;**

The parser converts this to an internal form. Upon execution, the execution engine evaluates the *expression* if it is present. If it is not, the value NULL is substituted. Then the current function terminates with that value as its apparent value in any expression it is embedded in. It is an error for there to be no enclosing function.

The *try* statement

The *try* statement has the form:

try *statement* **onerror** *statement*

The parser converts this to an internal form. Upon execution, the first *statement* is executed. If this statement executes normally flow continues after the *try* statement; the second *statement* is ignored. But if an error occurs during the execution of the first *statement* control is passed immediately to the second *statement*.

Note that "during the execution" applies to any depth of function calls, even to other modules or the parsing of sub-modules. When an error occurs both the parser and execution engine unwind as necessary until an error catcher (that is, a *try* statement) is found.

Errors can occur almost anywhere and for a variety of reasons. They can be explicitly generated with the *fail* function (described below), they can be generated as a side-effect of execution (such as division by zero), and they can be generated by the parser due to syntax or semantic errors in the parsed source. For whatever reason an error is generated, a message (a string) is always associated with it.

When any otherwise uncaught error occurs during the execution of the first *statement*, two things are done:

- Firstly, the string associated with the failure is assigned to the variable *error*. The assignment is made as if by a simple assignment statement within the scope of the *try* statement.
- Secondly, flow of control is passed to the statement following the *onerror* keyword.

Once the second *statement* finishes execution, flow of control continues as if the whole *try* statement had executed normally.

For example:

```
static
div(a, b)
{
    try
        return a / b;
    onerror
        return 0;
}

printf("4 / 2 = %d\n", div(4, 2));
printf("4 / 0 = %d\n", div(4, 0));
```

When run will print:

```
4 / 2 = 2
4 / 0 = 0
```

The handling of errors which are not caught by any *try* statement is implementation dependent. A typical action is to prepend the file and line number on which the error occurred to the error string, print this, and exit.

The null statement

The null statement has the form:

```
                    ;
```

The parser may convert this to an internal form. Upon execution it will do nothing.

Declaration statements

There are two types of declaration statements:

declaration	*storage-class declaration-list* `;`
	storage-class identifier function-body
storage-class	**extern**
	`static`
	`auto`

The first is the general case while the second is an abbreviated form for function definitions. Declaration statements are syntactically equal to any other statement, but their effect is made entirely at parse time. They act as null statements to the execution engine. There are no restriction on where they may occur, but their effect is a by-product of their parsing, not of any execution.

Declaration statements must start with one of the *storage-class* keywords listed above[3]. Considering the general case first, we next have a *declaration-list.*

declaration-list	*identifier [* `=` *expression]*
	declaration-list `,` *identifier [* `=` *expression]*

That is, a comma separated list of identifiers, each with an optional initialisation, terminated by a semicolon. For example:

```
    static a, b = 2, c = [array 1, 2, 3];
```

The storage class keyword establishes which scope the variables in the list are established in, as discussed earlier. Note that declaring the same identifier at different scope levels is permissible and that they are different variables.

A declaration with no initialisation first checks if the variable already exists at the given scope. If it does, it is left unmodified. In particular, any value it currently has is undisturbed. If it does not exist it is established and is given the value NULL.

A declaration with an initialisation establishes the variable in the given scope and gives it the given value even if it already exists and even if it has some other value.

Note that initial values are parser evaluated expressions. That is they are evaluated immediately by the parser, but may take arbitrary actions apart from that. For example:

```
    static
    fibonacci(n)
```

3.Note that, unlike C, function definitions must be prefixed by a storage class. As executable code may occur anywhere, this is required to distinguish them from a function call.

```
    {
        if (n <= 1)
            return 1;
        return fibonacci(n - 1) + fibonacci(n - 2);
    }

    static fib10 = fibonacci(10);
```

The declaration of *fib10* calls a function. But that function has already been defined so this will work.

Note that the scope of a static variable is (normally) the entire module it is parsed in. For example:

```
    static
    func()
    {
        static aStatic = "The value of a static.";
    }

    printf("%s\n", aStatic);
```

when run will print:

```
    The value of a static.
```

That is, despite being declared within a function, the declaration of *aStatic* has the same effect as if it had been declared outside the function. Also notice that the function has not been called. The act of parsing the function caused the declaration to take effect.

The behavior of extern variables has already been discussed, that is, they are declared as static in the parent module. The behavior of auto variables, and in particular their initialisation, will be discussed in a later section.

Abbreviated function declarations

As seen above there are two forms of declaration. The second:

storage-class identifier function-body

is a shorthand for:

storage-class identifier **`= [ func`** *function-body* **`] ;`**

and is the normal way to declare simple functions. Examples of this have been seen above.

Functions

As with most ICI constructs there are two parts to understanding functions; how they are parsed and how they execute.

When a function is parsed four things are noted:

- the names and positions of the formal parameters;
- the names and initialisation of auto variables;
- the static scope in which the function is declared;
- the code generated by the statements in the function.

The formal parameters (that is, the identifiers in the bracket enclosed list just before the compound statement) are actually implicit auto variable declarations. Each of the identifiers is declared as an auto variable without an initialisation, but in addition, its name and position in the list is noted.

Upon execution (that is, upon a function call), the following takes place:

- The auto variables, as noted by the parser, along with any initialisations, are copied as a group. This copy forms the auto variables of this invocation.
- Any actual parameters (that is, expressions provided by the caller) are matched positionally with the formal parameter names, and the value of those expressions are assigned to the auto variables of those names.
- If there were more actual parameters than formal parameters, and there is an auto variable called *vargs*, the remaining argument values are formed into an array which is assigned to *vargs*.
- The variable scope is set such that the auto variables are the inner-most scope, the static variables noted with the function are the next outer scope etc.
- The flow of control is diverted to the code generated by parsing the function.

A *return* statement executed within the function will cause the function to return to the caller and act as though its value were the expression given in the return statement. If no expression was given in the return statement, or if execution fell through the bottom of the function, the apparent return value is NULL. In any event, upon return the scope is restored to that of the caller. All internal references to the group of automatic variables are lost (although as will be seen later explicit program references may cause them to remain active).

Simple functions have been seen in earlier examples. We will now consider further issues.

It is very important to note that the parser generates a prototype set of auto variables which are copied, along with their initial values, when the function is called. The value which an auto variable is initialised with is a parser evaluated expression just like any other initialisation. It is not evaluated on function entry. But on function entry the value the parser determined is used to initialise the variable. For example:

```
static myVar = 100;

static
myFunc()
{
    auto anAuto = myVar;
```

```
        printf("%d\n", anAuto);
        anAuto = 500;
    }

    myFunc();
    myVar = 200;
    myFunc();
```

When run will print:

```
    100
    100
```

Notice that the initial value of *anAuto* was computed just once, changing *myVar* before the second call did not affect it. Also note that changing *anAuto* during the function did not affect its subsequent re-initialisation on the next invocation.

As stated above, formal parameters are actually uninitialised auto variables. Because of the behavior of variable declarations it is possible to explicitly declare an auto variable as well as include it in the formal parameter list. In addition, such an explicit declaration may have an initialisation. In this case, the explicit initialisation will be effective when there is no actual parameter to override it. For example:

```
    static
    print(msg, file)
    {
        auto file = stdout; /* Default value. */

        fprintf(file, "%s\n", msg);
    }

    print("Hello world");
    print("Hello world", stderr);
```

In the first call to the function *print* there is no second actual parameter. In this case the explicit initialisation of the auto variable *file* (which is the second formal parameter) will have its effect unmolested. But in the second call to *print* a second argument is given. In this case this value will over-write the explicit initialisation given to the argument and cause the output to go to *stderr*.

As indicated above there is a mechanism to capture additional actual parameters which were not mentioned in the formal parameter list. Consider the following example:

```
    static
    sum()
    {
        auto vargs;
        auto total = 0;
        auto arg;
        forall (arg in vargs)
            total += arg;
        return total;
    }

    printf("1+2+3 = %d\n", sum(1, 2, 3));
    printf("1+2+3+4 = %d\n", sum(1, 2, 3, 4));
```

Which when run will produce:

```
    1+2+3 = 6
    1+2+3+4 = 10
```

In this example the unmatched actual parameters were formed into an array and assigned to the auto variable *vargs*, a name which is recognised specially by the function call mechanism.

And also consider the following example where a default initialisation to *vargs* is made. In the following example the function *call* is used to invoke a function with an array of actual parameters, the function *array* is used to form an array at run-time, and addition is used to concatenate arrays; all these features will be further explained in later sections:

```
    static
    debug(fmt)
    {
        auto fmt = "Reached here.\n";
        auto vargs = [array];

        call(fprintf, array(stderr, fmt) + vargs);
    }

    debug();
    debug("Done that.\n");
    debug("Result = %d, total = %d.\n", 123, 456);
```

When run will print:

```
    Reached here.
    Done that.
    Result = 123, total = 456.
```

In the first call to *debug* no arguments are given and both explicit initialisations take effect. In the second call the first argument is given, but the initialisation of *vargs* still takes effect. But in the third call there are unmatched actual parameters, so these are formed into an array and assigned to *vargs*, overriding its explicit initialisation.

Method **Calls**

In addition to the above ICI has a simple mechanism for calling *methods* — functions contained within an object (typically a *struct*) that accept that object as their first parameter. The method call mechanism is enabled via a modification to the *call* operator, "()", to add semantics for calling a pointer object and through the addition of a new operator, binary-@, to form a pointer object from an object and a key. ICI pointers, described below, consist of an object and a key. To indirect though the pointer the object is indexed by the key and the resulting object used as the result. This is the same operation used in dynamic dispatch in languages such as Smalltalk and Objective-C.

The call operator now accepts a pointer as its first operand (we may think of the call operator as a n-ary operator that takes a function or pointer object as its first operand the function parameters as the remaining operands). When a pointer is "called" the key is used to index the pointer’s container object and the result, which must be a function object, is called. In addition the container object within the pointer is passed as an implicit first parameter to the function (thus passing the actual object used to invoke the method to the method). Apart from the calling semantics the functions used to implemented methods are in all respects normal ICI functions.

Struct objects are typically used as the "container" for objects used with methods. The super mechanism provides the hierarichal search needed to allow class objects to be shared by multiple instances and provide a natural means of encapsulating information.

A typical way of using methods is,

```
/*
 * Define a "class" object representing our class and
 * containing the class methods.
 */
static MyClass = [struct

   doubleX = [func (self)
   {
      return self.x * 2;
   }]

];

...

static a;
a = struct(@MyClass);
a.x = 21;
printf("%d\n", a@doubleX());
```

We first define a class by using a literal struct to contain our named methods. You could also define class variables in this struct as it is shared by all instances of that class. In our class we’ve got a single method, doubleX, that doubles the value of an instance variable called x.

Later in the program we create an instance of a MyClass object by making a new struct object and setting its super struct to the class struct. The super is made atomic which ensures all instances share the same object and makes it read-only for them. Then we create an "instance variable" within the object by assigning 21 to a.x and finally invoke the method. We do not pass any parameters to doubleX. The call through the pointer object formed by the binary-@ operator passes "a" implicitly

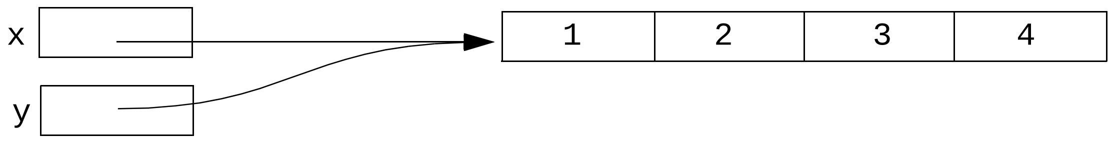
Objects

Up till now few exact statements about the nature of values and data have been made. We will now examine values in more detail. Consider the following code fragment:

```
    static x;
    static y;

    x = [array 1, 2, 3, 4];
    y = x;
```

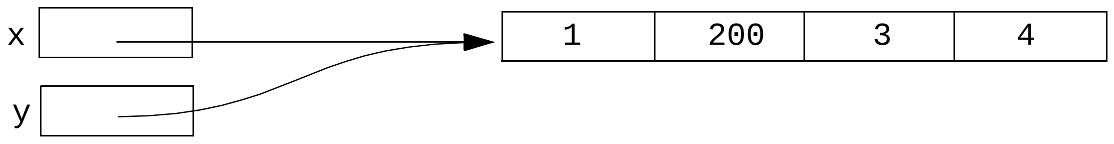
After execution of this code the variable *x* refers to an array. The assignment of *x* to *y* causes *y* to refer to the same array. Diagrammatically:



If the assignment:

```
    y[1] = 200;
```

is performed, the result is:

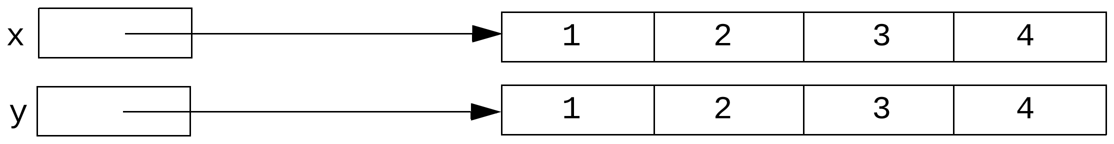


We say that *x* and *y* refer to the same object. Now consider the following code fragment:

```
    static x;
    static y;

    x = [array 1, 2, 3, 4];
    y = [array 1, 2, 3, 4];
```

Diagrammatically:

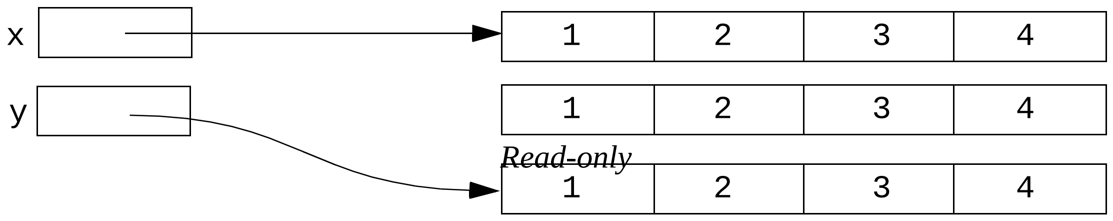


In this case, *x* and *y* refer to different objects, despite that fact they are equal.

Now consider one of the unary operators which was only briefly mentioned in the sections above. The @ operator returns a read-only version of the sub-expression it is applied to. Consider the following statement:

```
    y = @y;
```

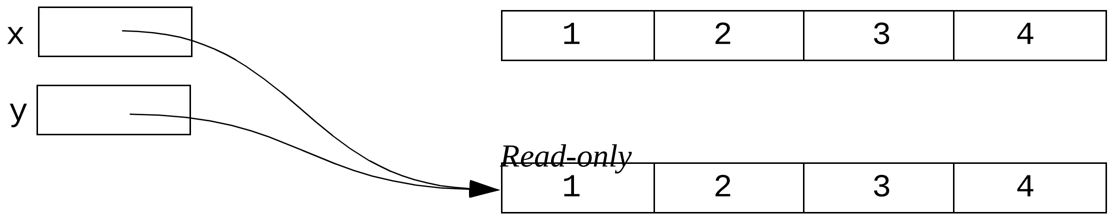
After this has been executed the result could be represented diagrammatically as:



The middle array now has no reference to it and the memory associated with it will be collected by the interpreter's standard garbage collection mechanism. Now consider the following statement:

```
    x = @x;
```

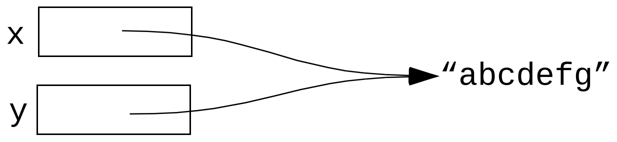
This is similar to the previous statement, except that this time *x* is replaced by a read-only version of its old value. But the result of this operation is:



Notice that *x* now refers to the same read-only array that *y* refers to. This is a fundamental property of the @ operator. It returns *the unique* read-only version of its argument value. Such read-only objects are referred to as *atomic* objects. The array which *x* used to refer to was non-atomic, but the array it refers to now is an atomic array. Aggregate types such as arrays, sets and structs are generally non-atomic, but atomic versions can be obtained (as seen above). But most other types, such as integers floats, strings and functions are intrinsically atomic. That is, no matter how a number, say 10, is generated, it will be the same object as every other number 10 in the interpreter. For-instance, consider the following example:

```
    x = "ab" + "cdefg";
    y = "abcde" + "fg";
```

After this is executed the situation can be represented diagrammatically as:



It is important to understand when objects are the same object, when they are different and the effects this has.

Equality

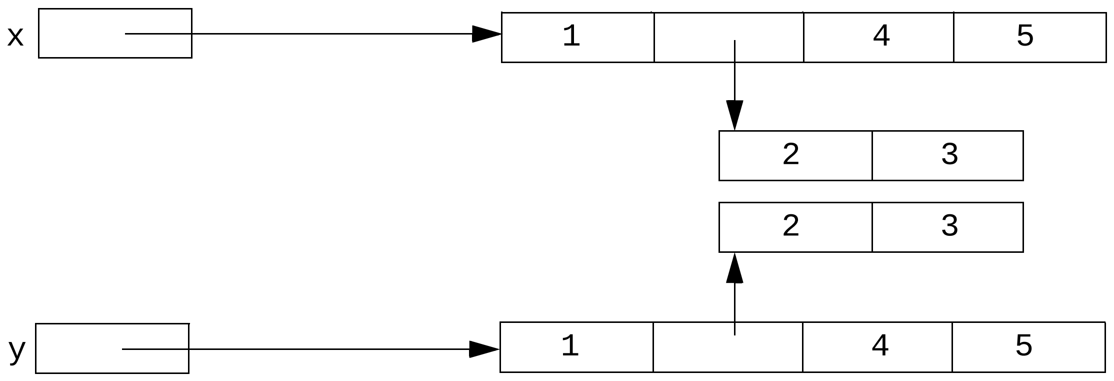
We saw above how two apparently identical arrays were each distinct object. But these two arrays were *equal* in the sense of the equality testing operator ==. If two values are the same object they are said to be *eq*[4], and there is a function of that name to test for this condition. Two objects are *equal* (that is ==) if:

- they are the same object; or
- they are both arithmetic (int and float) and have equivalent numeric values; or
- they are aggregates of the same type and all the sub-elements are the same objects.

This definition of equality is the basis for resolving the merging of aggregates into unique read-only (atomic) versions. Two aggregates will resolve to the same atomic object if they are *equal*. That is, they must contain exactly the same objects as sub-elements, not just equal objects. For example:

```
    static x = [array 1, [array 2, 3], 4, 5];
    static y = [array 1, [array 2, 3], 4, 5];
```

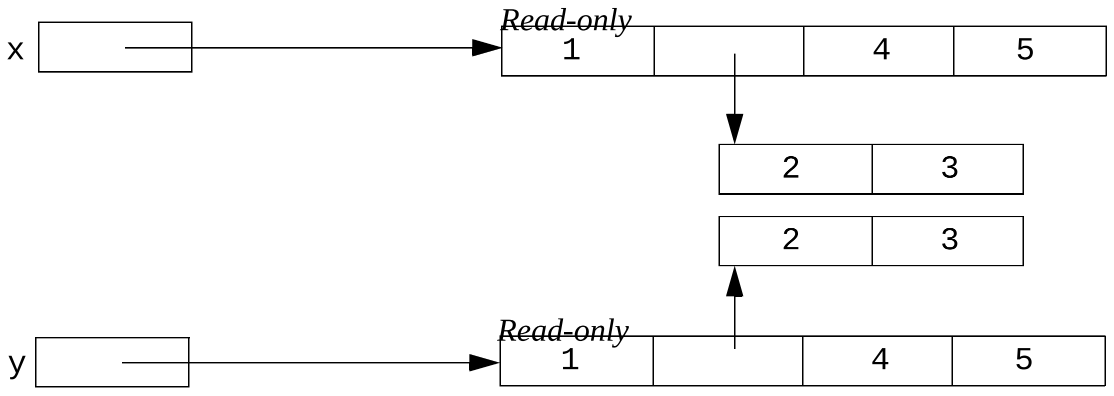
Could be represented diagrammatically as:



Now, if the following statements were executed:

```
    x = @x;
    y = @y;
```

The result could be represented diagrammatically as:

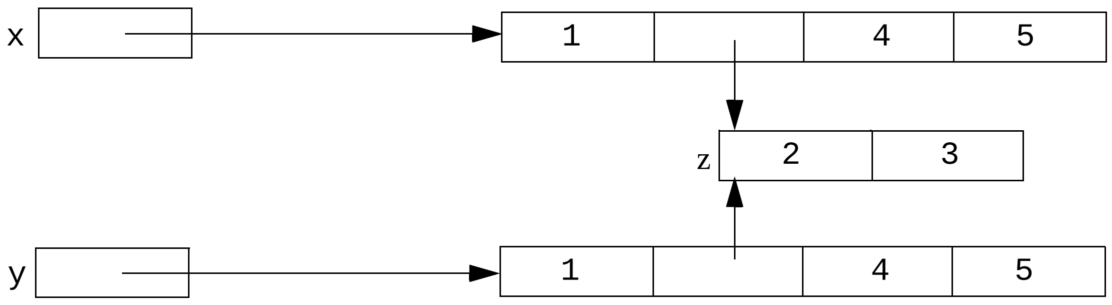


4.As in LISP.

That is, both *x* and *y* refer to new read-only objects, but they refer to different read-only objects because they have an element which is not the same object. The simple integers are the same objects because integers are intrinsically atomic objects. But the two sub-arrays are distinct objects. Being equal was not sufficient. The top-level arrays needed to have exactly the same objects as contents to make *x* and *y* end up referring to the same read-only array. In contrast to this consider the following similar situation:

```
static z = [array 2, 3];
static x = [array 1, z, 4, 5];
static y = [array 1, z, 4, 5];
```

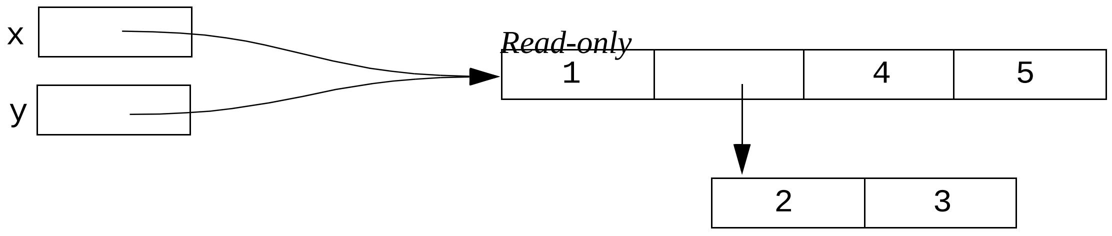
This could be represented diagrammatically as:



Now, if the following statements were executed:

```
x = @x;
y = @y;
```

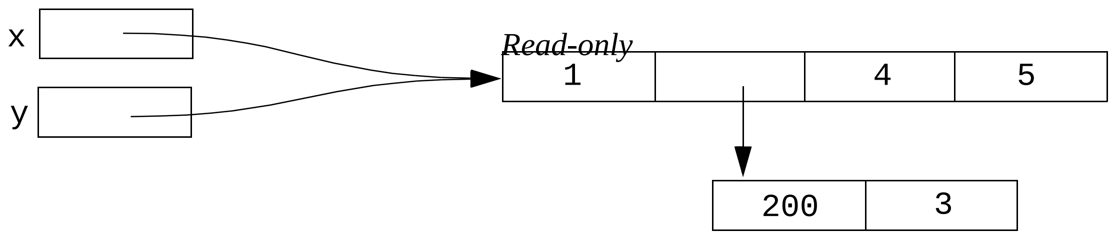
The result could be represented diagrammatically as:



In this case both *x* and *y* refer to the same read-only array because the original arrays where equal, that is, all their elements were the same objects. Notice that one of the elements is still a *writeable* array. The read-only property is only referring to the top level array. The sub-array can be changed, but the reference to it from the top level array can not. Thus:

```
x[1][0] = 200;
```

will result in:



whereas the statement:

```
    x[1] = 200;
```

will just result in an error.

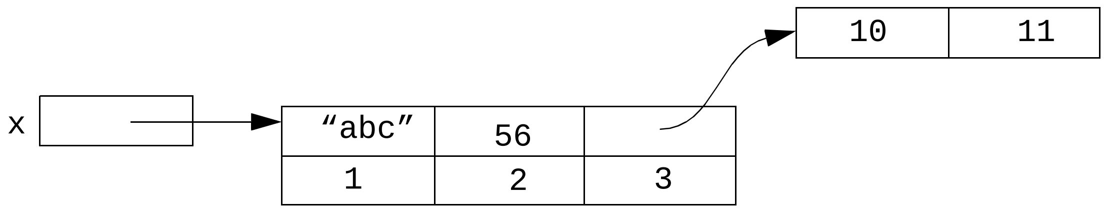
Structure and set keys

Any object, not just a string, can be used as a key in a structure. For instance:

```
    static x = [struct];
    static z = [array 10, 11];

    x["abc"] = 1;
    x[56] = 2;
    x[z] = 3;
```

Could be represented diagrammatically as:



And the assignment:

```
    x[z] = 300;
```

would replace the *3* in the above diagram with *300*. But the assignment:

```
    x[[array 10, 11]] = 300;
```

would result in a new element being added to the structure because the array given in the above statement is a different object from the one which *z* refers to.

Similarly, elements of sets may be any objects.

Indexing structures by complex aggregates is as efficient as indexing by intrinsically atomic types such as strings and integers.

Structure super types

Up till now structures have been described as simple lookup tables which map a key, or index, to a value. But a structure may have associated with it a *super structure*.

The function *super* can be used to discover the current super of a struct and to set a new super. With just one argument it returns the current super of that struct, with a second argument it also replaces the super by that value.

When a key is being looked-up in a structure for reading, and it is not found and there is a *super struct*, the key is further looked for in the super struct, if it is found there its value from that struct is returned. If it is not found it will be looked for in the next super struct etc. If no structures in the *super chain* contain the key, the special value NULL is returned.

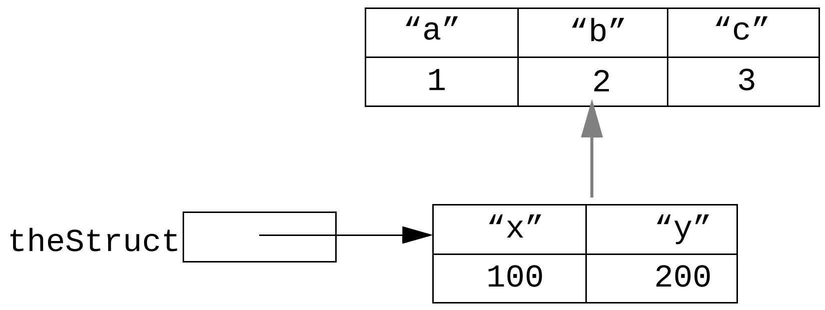
When a key is being looked up in a structure for writing, it will similarly be searched for in the super chain. If it is found in a writeable structure the value in the structure in which it was found will be set to the new value. If it was never found, it will be added along with the given value to the very first struct, that is, the structure at the base, or root, of the super chain.

Consider the following example:

```
    static theSuper = [struct a = 1, b = 2, c = 3];
    static theStruct = [struct x = 100, y = 200];

    super(theStruct, theSuper);
```

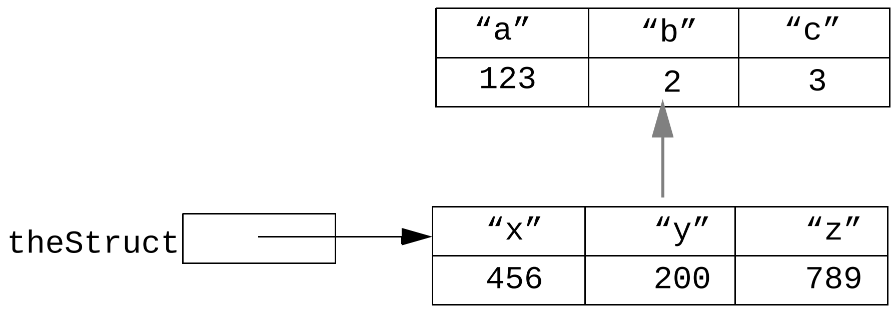
After this statement the situation could be represented diagrammatically as:



then if the following statements were executed:

```
    theStruct.a = 123;
    theStruct.x = 456;
    theStruct.z = 789;
```

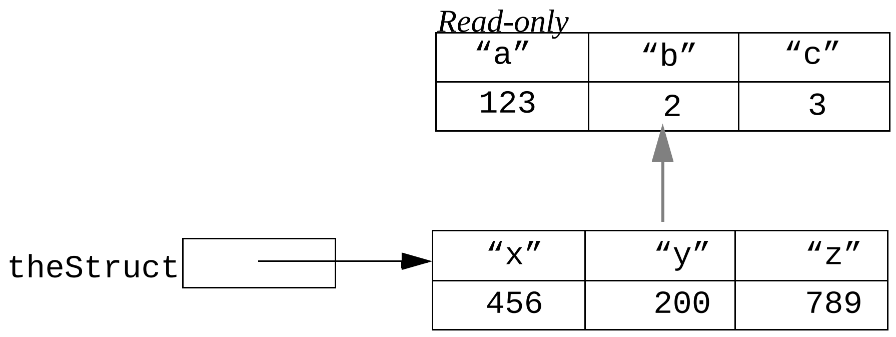
the situation could be diagrammatically represented as:



If a super struct is not writeable (that is, it is atomic) values will not be written in it and will lodge in the base structure instead. Thus consider what happens if we replace the super structure in the previous example by its read-only version:

```
    super(theStruct, @theSuper);
```

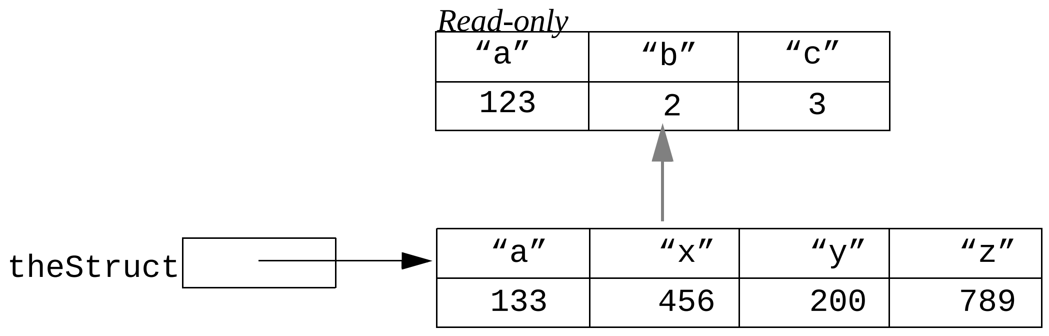
The situation could now be represented diagrammatically as:



If the assignment statement:

```
    theStruct.a += 10;
```

were executed, the value of the element *a* will first be *read* from the super structure, this value will then have ten added to it, and the result will be *written* back into the base structure; because the super structure is read-only and cannot be modified. The finally situation can be represented diagrammatically as:



Note that many structs may share the same super struct. Thus a single read-only super struct can be used hold initial values; saving explicit initialisations and storage space.

The function *assign* may be used to set a value in a struct explicitly, without reference to any super structs; and the function *fetch* may be used to read a value from a struct explicitly, without reference to any super structs.

Within a *struct-literal* a colon prefixed expression after the *struct* identifier is used as the super struct. For example, the declarations used in the previous example could be written as:

```
    static theSuper = [struct a = 1, b = 2, c = 3];
    static theStruct = [struct:theSuper, x = 100, y = 200];
```

An aside on variables and scope

Now that structs and their super have been described a more precise statement about variables and scope can be made.

ICI variables are entries in ordinary structs. At all times, the current scope is identified by a structure. The auto variables are the entries in this base structure. Its super is the struct containing the static variables. The next super struct contains the externs, and successive super structs are successive outer scopes.

Auto, static and extern declarations make explicit assignments to the appropriate structure.

In these terms it can be said that an un-adorned identifier in an expression is an implicit reference to an element of the current scope structure. The inheritance and name hiding of the variable scope mechanism is a product of the super chain. But there is a difference in the handling of undefined entries. Whereas normal lookup of undefined entries in a structure produces a default value of NULL or implicit creation, the implicit lookup of undefined variables triggers an attempt to dynamically load a library to define the variable (see *Undefined variables and dynamic loading* below), and failing that, produce an error ("%s undefined").

The function *scope* can be used to obtain the current scope structure; and to set it (use with care).

Note that when there is an atomic structure in the scope chain the mechanism described at the end of the previous section does not operate correctly. Writing to a variable in the atomic struct will give a spurious undefined error rather than lodging it in the base structure. This is a deficiency which will be corrected in a later release.

Pointers

Pointers are references to storage locations. Storage locations are the elements of anything which can be indexed. That is, array elements, set elements, struct elements and others (which we will see below) can be pointed to. Variables (which are just struct elements) can be pointed to. In more general terms, any lvalue can be pointed to.

The *&* operator is used to obtain a pointer to a location. Thus if the following were executed:

```
    static x;
    static y = [array 1, 2, 3];
    static p1 = &x;
    static p2 = &y[1];
```

The variable *p1* would be a pointer to *x* and the variables *p2* would be a pointer to the second element of *y*. Reference to the object a pointer points to can be obtained with the * operator. Thus if the following were executed:

```
    *p1 = 123;
    *p2 = 456;
    printf("x = %d, y[1] = %d\n", x, y[1]);
```

the output would be:

```
    x = 123, y[1] = 456
```

Pointers are really a bundle of two objects, one is the object pointed into, the other is the key used to access the location in question. For instance, in the example above *p2* remembers the array, and the number 1; that is, the aggregate and the index. The generation of a pointer does not affect the location being pointed to. In fact the location may not even exist yet. When a pointer is referenced the same operation takes place as if the location was referenced explicitly. Thus a search down the super chain of a struct may occur, or an array may be extended to include the index being written to, etc.

In addition to simple indirection (that is the * operator), pointers may be indexed. But the index values must be an integer, and the key stored as part of the pointer must also be an integer. When a pointer is indexed, the index is added to the key which is stored as part of the pointer, the sum forms the actual index to use to when referencing the aggregate recorded by the pointer. For instance, continuing the example above:

```
    p2[1] = 789;
```

would set the last element of the array to 789, because the pointer currently references element 1, and the given index is 1, and 1 + 1 is 2 which is the last element. The index arithmetic provided by pointers will work with any types, as long as the indexes are integers, thus:

```
    static s = [struct (20) = 1, (30) = 2, (40) = 3];
    static p = &s[30];

    p[-10] = -1;
    p[0] = -2;
    p[10] = -3;
```

Would replace each of the elements in the struct *s* by their negative value.

This concludes the general discussion of ICI as a whole. We will now examine the exact nature of each of the data types, then each of the expression operators, and finally each of the standard functions.

Data types

ICI supports a base set of standard data types. Each is identified by a simple name. In summary these are:

array	An ordered sequence of other objects.
file	An open file reference.
float	A double precision floating point number.
func	A function.
int	A signed 32 bit integer.
mem	References to raw machine memory.
ptr	A reference to a storage location.
regexp	A compiled regular expression.
set	An unordered collection of other objects.
string	An ordered sequence of 8 bit characters.
struct	An unordered set of pairs of objects.

A full explanation of the semantics of each type (including the semantics of indexing an object of that type) will be included in a future version of this document.

Operators

The following table details each of the unary and binary operators with all of the types they may be applied to. Within the first column the standard type names are used to stand for operands of that type, along with *any* to mean any type and *num* to mean an *int* or a *float*. In general, where an *int* and a *float* are combined in an arithmetic operation, the *int* is first converted to a *float* and then the operation is performed.

The following table is in precedence order.

*****_ptr_ Indirection: The result references the thing the pointer points to. The result is an lvalue.

&_any_ Address of: The result is a pointer to *any*. If *any* is an lvalue the pointer references that storage location. If *any* is not an lvalue but is a *term* other than a bracketed non-*term*, as described in the syntax above, a one element array containing *any* will be fabricated and a pointer to that storage location returned. For example:

```
p = &1;
```

sets p to be a pointer to the first element of an un-named array, which currently contains the number 1.

-_num_ Negation: Returns the negation of *num*. The result is the same type as the argument. The result is not an lvalue.

+_any_ Has no effect except the result is not an lvalue.

!_any_ Logical negation: If *any* is 0 (integer) or NULL, 1 is returned, else 0 is returned.

~*int*	Bit-wise complement: The bit-wise complement of *int* is returned.
++*any*	Pre-increment: Equivalent to **(***any* **+= 1)**. *any* must be an lvalue and obey the restrictions of the binary **+** operator. See **+** below.
--*any*	Pre-decrement: Equivalent to **(***any* **-= 1)**. *any* must be an lvalue and obey the restrictions of the binary **-** operator. See **-** below.
@*any*	Atomic form of: Returns the unique, read-only form of *any*. If *any* is already atomic, it is returned immediately. Otherwise an atomic form of *any* is found or generated and returned; this is of execution time order equal to the number of elements in *any*. See the section on objects above for more explanation.
$*any*	Immediate evaluation: Recognised by the parser. The sub-expression *any* is immediately evaluated by invocation of the execution engine. The result of the evaluation is substituted directly for this expression term by the parser.
any++	Post-increment: Notes the value of any, then performs the equivalent of **(***any* += **1)**, except any is only evaluated once, and finally returns the original noted value. any must be an lvalue and obey the restrictions of the binary + operator. See + below.
any--	Post-increment: Notes the value of any, then performs the equivalent of **(***any* -= **1)**, except any is only evaluated once, and finally returns the original noted value. any must be an lvalue and obey the restrictions of the binary - operator. See - below.
any1 @ any2	*Form pointer: Returns a pointer object formed from its operands with the pointer's aggregate being set from any1 and the pointer's key from any2.*
num1 ***** *num2*	Multiplication: Returns the product of the two numbers, if both nums are ints, the result is int, else the result is float.
set1 ***** *set2*	Set intersection: Returns a set that contains all elements that appear in both *set1* and *set2*.
num1 / *num2*	Division: Returns the result of dividing *num1* by *num2*. If both numbers are ints the result is int, else the result is float. If *num2* is zero the error *division by 0* is generated, or *division by 0.0* if the result would have been a float.
int1 **%** *int2*	Modulus: Returns the remainder of dividing *int1* by *int2*. If *int2* is zero the error *modulus by 0* is generated.
num1 **+** *num2*	Addition: Returns the sum of *num1* and *num2*. If both numbers are *ints* the result is *int*, else the result is *float*.

ptr + *int*	Pointer addition: *ptr* must point to an element of an indexable object whose index is an *int*. Returns a new pointer which points to an element of the same aggregate which has the index which is the sum of *ptr*'s index and *int*. The arguments may be in any order.
string1 + *string2*	String concatenation: Returns the string which is the concatenation of the characters of *string1* then *string2*. The execution time order is proportional to the total length of the result.

array1 + *array2*	Array concatenation: Returns a new array which is the concatenation of the elements from *array1* then *array2*. The execution time order is proportional to the total length of the result. Note the difference between the following:
	
	`a += [array 1];`
	`push(a, 1);`
	
	In the first case a is replaced by a newly formed array which is the original array with one element added. But in the second case the *push* function (see below) appends an element to the array *a* refers to, without making a new array. The second case is much faster, but modifies an existing array.

struct1 + *struct2*	Structure concatenation: Returns a new struct which is a copy of *struct1*, with all the elements of *struct2* assigned into it. Obeys the semantics of copying and assignment discussed in other sections with regard to super structs.. The execution time order is proportional to the sum of the lengths of the two arguments.
set1 + *set2*	Set union: Returns a new set which contains all the elements from both sets. The execution time order is proportional to the sum of the lengths of the two arguments.
num1 - *num2*	Subtraction: Returns the result of subtracting *num2* from *num1*. If both numbers are ints the result is *int*, else the result is *float*.
set1 - *set2*	Set subtraction: Returns a new set which contains all the elements of *set1*, less the elements of *set2*. The execution time order is proportional to the sum of the lengths of the two arguments.
ptr1 - *ptr2*	Pointer subtraction: *ptr1* and *ptr2* must point to elements of indexable objects whose indexs are *ints*. Returns an *int* which is the the index of *ptr1* less the index of *ptr2*.
int1 >> *int2*	Right shift: Returns the result of right shifting *int1* by *int2*. Equivalent to division by 2***int2*. *int1* is interpreted as a signed quantity.
int1 << *int2*	Left shift: Returns the result of left shifting *int1* by *int2*. Equivalent to multiplication by 2***int2*.

array << *int*	Left shift array: Returns a new array which contains the elements of *array* from index *int* onwards. Equivalent to the function call *interval(array, int)* (which is considered preferable, this operator may disappear in future releases).
num1 < *num2*	Numeric test for less than: Returns 1 if *num1* is less than *num2,* else 0.
set1 < *set2*	Test for subset: Returns 1 if *set1* contains only elements that are in *set2,* else 0.
string1 < *string2*	Lexical test for less than: Returns 1 if *string1* is lexically less than *string2,* else 0.
ptr1 < *ptr2*	Pointer test for less than: *ptr1* and *ptr2* must point to elements of indexable objects whose indexes are *ints*. Returns 1 if *ptr1* points to an element with a lesser index than *ptr2,* else 0.

The >, <= and >= operators work in the same fashion as <, above. For sets > tests for one set being a superset of the other. The <= and >= operators test for proper sub- or supersets. That is one set can contain only those elements contained in the other set but cannot be equal to the other set.

any1 == *any2*	Equality test: Returns 1 if *any1* is equal to *any2,* else 0. Two objects are equal when: they are the same object; or they are both arithmetic (*int* and *float*) and have equivalent numeric values; or they are aggregates of the same type and all the sub-elements are the same objects.
any1 **!=** *any2*	Inequality test: Returns 1 if *any1* is not equal to *any2,* else 0. See above.
string ~ *regexp*	Logical test for regular expression match: Returns 1 if *string* can be matched by *regexp,* else 0. The arguments may be in any order.
string **!~** *regexp*	Logical test for regular expression non-match: Returns 1 if *string* can not be matched by *regexp,* else 0. The arguments may be in any order.
string ~~ *regexp*	Regular expression sub-string extraction: Returns the sub-string of *string* which is matched by the first bracket enclosed portion of *regexp,* or NULL if there is no match or *regexp* does not contain a (...) portion. The arguments may be in any order. For example, a "basename" operation can be performed with:

```
argv[0] ~~= #([^/]*)$#;
```

string ~~~ *regexp*	Regular expression multiple sub-string extraction: Returns an array of the the sub-strings of *string* which are matched by the (...) enclosed portions of *regexp*, or NULL if there is no match. The arguments may be in any order.
int1 **&** *int2*	Bit-wise and: Returns the bit-wise and of *int1* and *int2*.
int1 ^ *int2*	Bit-exclusive or: Returns the bit-wise exclusive or of *int1* and *int2*.
int1 | *int2*	Bit-wise or: Returns the bit-wise or of *int1* and *int2*.
any1 **&&** *any2*	Logical and: Evaluates the expression which determines *any1*, if this evaluates to 0 or NULL (i.e. *false*), 0 is returned, else *any2* is evaluated and returned[5]. Note that if *any1* does not evaluate to a *true* value, the expression which determines *any2* is never evaluated.
any1 || *any2*	Logical or: Evaluates the expression which determines *any1*, if this evaluates to other than 0 or NULL (i.e. *true*), 1 is returned, else *any2* is evaluated and returned. Note that if *any1* does not evaluate to a *false* value, the expression which determines *any2* is never evaluated.
any1 **?** *any2* **:** *any3*	Choice: If *any1* is neither 0 or NULL (i.e. *true*), the expression which determines *any2* is evaluated and returned, else the expression which determines *any3* is evaluated and returned. Only one of *any2* and *any3* are evaluated. The result may be an lvalue if the returned expression is. Thus: `flag ? a : b = value` is a legal expression and will assign *value* to either *a* or *b* depending on the state of *flag*.
any1 = *any2*	Assignment: Assigns *any2* to *any1*. *any1* must be an lvalue. The behavior of assignment is a consequence of aggregate access as discussed in earlier sections. In short, an lvalue (in this case *any1*) can always be resolved into an aggregate and an index into the aggregate. Assignment sets the element of the aggregate identified by the index to *any2*. The returned result of the whole assignment is *any1*, after the assignment has been performed. The result is an lvalue, thus: `++(a = b)` will assign *b* to *a* and then increment *a* by 1.

5.Note that this is different from C where the result is always completely resolved to a 0 or 1. Use !! to force a 0/1 value from a generic true/false.

Note that assignment operators (this and following ones) associate right to left, unlike all other binary operators, thus:

```
a = b += c -= d
```

Will subtract *d* from *c*, then add the result to *b*, then assign the final value to *a*.

`+= -= *= /= %= >>= <<= &= ^= |= ~~=`

Compound assignments: All these operators are defined by the rewriting rule:

any1 op= any2

is equivalent to:

any1 = any1 op any2

except that *any1* is not evaluated twice. Type restrictions and the behavior or *op* will follow the rules given with that binary operator above. The result will be an lvalue (as a consequence of = above). There are no further restrictions. Thus:

```
a = "Hello";
a += " world.\n";
```

will result in the variable *a* referring to the string:

```
"Hello world.\n".
```

any1 <=> *any2* Swap: Swaps the current values of *any1* and *any2*. Both operands must be lvalues. The result is *any1* after the swap and is an lvalue, as in other assignment operators. Also like other assignment operators, associativity is right to left, thus:

```
a <=> b <=> c <=> d
```

rotates the values of *a*, *b* and *c* towards *d* and brings *d*'s original value back to *a*.

any1 **,** *any2* Sequential evaluation: Evaluates *any1*, then *any2*. The result is *any2* and is an lvalue if *any2* is. Note that in situations where comma has meaning at the top level of parsing an expression (such as in function call arguments), expression parsing precedence starts at one level below the comma, and a comma will not be recognised as an operator. Surround the expression with brackets to avoid this if necessary.

Core language functions

The following list summarises the standard functions. Following this is a detailed descriptions of each of them.

```
     float|int =abs(float|int)
         float =acos(number)
           mem =alloc(int [, int])
         array =array(any...)
         float =asin(number)
           any =assign(struct, any, any)
         float =atan(number)
         float =atan2(number, number)
           any =call(func, array)
         float =ceil(number)
                close(file)
           any =copy(any)
         float =cos(number)
          file =currentfile()
                del(struct, any)
           int =eq(any, any)
           int =eof(file)
                eventloop()
                exit([int|string|NULL])
         float =exp(number)
         array =explode(string)
                fail(string)
           any =fetch(struct, any)
         float =float(any)
         float =floor(number)
           int =flush(file)
         float =fmod(number, number)
          file =fopen(string [, string])
                flush([file])
        string =getchar([file])
        string =getfile([file])
        string =getline([file])
        string =getenv(string)
        string =gettoken([file|string [,string]])
         array =gettokens([file|string [,string [,string]]])

        string =gsub(string, regexp, string)
        string =implode(array)
        struct =include(string [, struct])
           int =int(any)
  string|array =interval(string|array, int [, int])
```

```
          int =isatom(any)
        array =keys(struct)
        float =log(number)
        float =log10(number)
          mem =mem(int, int [,int])
         file =mopen(string [, string])
          int =nels(any)
    int|float =num(string|int|float)
       struct =parse(file|string [, struct])
          any =pop(array)
         file =popen(string [, string])
        float =pow(number, number)
               printf([file,] string [, any...])
          any =push(array, any)
               put(string)
               putenv(string [, string])
          int =rand([int])
               reclaim()
       regexp =regexp(string)
       regexp =regexpi(string)
               remove(string)
       struct =scope([struct])
          int =seek(file, int, int)
          set =set(any...)
        float =sin(number)
          int =sizeof(any)
        array =smash(string, string)
         file =sopen(string [, string])
               sort(array, func)
       string =sprintf(string [, any...])
        float =sqrt(number)
       string =string(any)
       struct =struct(any, any...)
       string =sub(string, regexp, string)
       struct =super(struct [, struct])
          int =system(string)
        float =tan(number)
       string =tochar(int)
          int =toint(string)
          any =top(array [, int])
          int =trace(string)
       string =typeof(any)
        array =vstack()
file|int|float =waitfor(file|int|float...)
```

The following is an alphabetic listing of each of the standard functions.

`float|int = abs(float|int)`

Returns the absolute value of its argument. The result is an int if the argument is an int, a float if it is a float.

`angle = acos(x)`

Returns the arc cosine of *x* in the range 0 to pi.

`mem = alloc(nwords [, wordz])`

Returns a new *mem* object referring to *nwords* (an int) of newly allocated and cleared memory. Each word is either 1, 2, or 4 bytes as specified by *wordz* (an int, default 1). Indexing of *mem* objects performs the obvious operations, and thus pointers work too.

`array = array(any...)`

Returns an array formed from all the arguments. For example:

```
    array()
```

will return a new empty array; and

```
    array(1, 2, "a string")
```

will return a new array with three elements, *1*, *2*, and *"the string"*.

This is the run-time equivalent of the array literal. Thus the following two expressions are equivalent:

```
    $array(1, 2, "a string")

    [array 1, 2, "a string"]
```

`float = asin(x)`

Returns the arc sine of *x* in the range -pi/2 to pi/2.

`value = assign(struct, key, value)`

Sets the element of *struct* identified by *key* to *value*, ignoring any super struct. Returns *value*.

`angle = atan(x)`

Returns the arc tangent of *x* in the range -pi/2 to pi/2.

`angle = atan2(y, x)`

Returns the angle from the origin to the rectangular coordinates *x, y* (floats) in the range -pi to pi.

`return = call(func, args)`

Calls the function *func* with arguments taken from the array *args*. Returns the return value of the function.

This is often used to pass on an unknown argument list. For example:

```
    static
    db()
    {
        auto vargs;

        if (debug)
            return call(printf, vargs);
    }
```

`new = copy(old)`

Returns a copy of *old*. If *old* is an intrinsically atomic type such as an int or string, the *new* will be the same object as the old. But if *old* is an array, set, or struct, a copy will be returned. The copy will be a new non-atomic object (even if *old* was atomic) which will contain exactly the same objects as *old* and will be *equal* to it (that is ==). If *old* is a struct with a super struct, *new* will have the same super (exactly the same super, not a copy of it).

`x = cos(angle)`

Returns the cosine of *angle* (a float interpreted in radians).

`file = currentfile()`

Returns the file associated with the innermost parsing context, or NULL if there is no module being parsed.

This function can be used to include data in a program source file which is out-of-band with respect to the normal parse stream. But to do this it is necessary to know up to what character in the file in question the parser has consumed.

In general: after having parsed any simple statement the parser will have consumed up to and including the terminating semicolon, and no more. Also, after having parsed a compound statement the parser will have consumed up to and including the terminating close brace and no more. For example:

```
    static help = gettokens(currentfile(), "", "!")[0]

    ;This is the text of the help message.
    It follows exactly after the ; because
    that is exactly up to where the parser
    will have consumed. We are using the
    gettokens() function (as described below)
    to read the text.
    !

    static otherVariable = "etc...";
```

This function can also be used to parse the rest of a module within an error catcher. For example:

```
    try
        parse(currentfile(), scope())
    onerror
        printf("That didn't work, but never mind.\n");

    static this = that;
    etc();
```

The functions *parse* and *scope* are described below.

del(struct, key)

Deletes the element of *struct* identified by *key*. Any super structs are ignored. Returns NULL. For example:

```
    static s = [struct a = 1, b = 2, c = 3];
    static v, k;
    forall (v, k in s)
        printf("%s=%d\n", k, v);
    del(s, "b");
    printf("\n");
    forall (v, k in s)
        printf("%s=%d\n", k, v);
```

When run would produce (possibly in some other order):

```
    a=1
    c=3
    b=2

    a=1
    c=3
```

int = eof([file])

Returns non-zero if end of file has been read on *file*. If *file* is not given the current value of *stdin* in the current scope is used.

int = eq(obj1, obj2)

Returns 1 (one) if *obj1* and *obj2* are the same object, else 0 (zero).

eventloop()

Enters an internal event loop and never returns (but can be broken out of with an error). The exact nature of the event loop is system specific. Some dynamically loaded modules require an event loop for their operation.

exit([string|int|NULL])

Causes the interpreter to finish execution and exit. If no parameter, the empty string or

NULL is passed the exit status is zero. If an integer is passed that is the exit status. If a non-empty string is passed then that string is printed to the interpreter's standard error output and an exit status of one used. This is implementation dependent and may be replaced by a more general exception mechanism. Avoid.

`float = exp(x)`

Returns the exponential function of *x*.

`array = explode(string)`

Returns an array containing each of the integer character codes of the characters in *string*.

`fail(string)`

Causes an error to be raised with the message *string* associated with it. See the section of error handling in the *try* statement above. For example:

```
    if (qf > 255)
        fail(sprintf("Q factor %d is too large", qf));
```

`value = fetch(struct, key)`

Returns the *value* from *struct* associated with *key*, ignoring any super structs. Returns NULL is *key* is not an element of *struct*.

`value = float(x)`

Returns a floating point interpretation of *x*, or 0.0 if no reasonable interpretation exists. *x* should be an int, a float, or a string, else 0.0 will be returned.

`file = fopen(name [, mode])`

Opens the named file for reading or writing according to *mode* and returns a file object that may be used to perform I/O on the file. *Mode* is the same as in C and is passed directly to the C library **`fopen`** function. If mode is not specified **`“r”`** is assumed.

`fprintf(file, fmt, args...)`

Formats a string based on *fmt* and *args* as per *sprintf* (below) and outputs the result to *file*. See *sprintf*. **Changes to ICI's printf have made fprintf redundant and it may be removed in future versions of the interpreter. Avoid.**

`string = getchar([file])`

Reads a single character from *file* and returns it as a string. Returns NULL upon end of file. If *file* is not given the current value of *stdin* in the current scope is used.

`string = getfile([file])`

Reads all remaining data from *file* and returns it as a string. Returns an empty string upon end of file. If *file* is not given the current value of *stdin* in the current scope is used.

`string = getline([file])`

Reads a line of text from *file* and returns it as a string. Any end-of-line marker is removed.

Returns *NULL* upon end of file. If *file* is not given the current value of *stdin* in the current scope is used.

`string = gettoken([file [, seps]])`

Read a token (that is, a string) from *file*.

Seps must be a string. It is interpreted as a set of characters which do not from part of the token. Any leading sequence of these characters is first skipped. Then a sequence of characters not in seps is gathered until end of file or a character from seps is found. This terminating character is not consumed. The gathered string is returned, or NULL if end of file was encountered before any token was gathered.

If *file* is not given the current value of *stdin* in the current scope is used.

If *seps* is not given the string " \t\n" is assumed.

`array = gettokens([file [, seps [, terms]]])`

Read tokens (that is, strings) from *file*. The tokens are character sequences separated by *seps* and terminated by *terms*. Returns an array of strings, NULL on end of file.

If *seps* is a string, it is interpreted as a set of characters, any sequence of which will separate one token from the next. In this case leading and trailing separators in the input stream are discarded.

If *seps* is an integer it is interpreted as a character code. Tokens are taken to be sequences of characters separated by exactly one of that character.

Terms must be a string. It is interpreted as a set of characters, any one of which will terminate the gathering of tokens. The character which terminated the gathering will be consumed.

If *file* is not given the current value of *stdin* in the current scope will be used.

If *seps* is not given the string " \t" is assumed.

If *terms* is not given the string "\n" is assumed.

For example:

```
forall (token in gettokens(currentfile()))
    printf("<%s>", token)
;   This    is my line    of data.
printf("\n");
```

when run will print:

```
<This><is><my><line><of><data.>
```

Whereas:

```
    forall (token in gettokens(currentfile(), ':', "*"))
        printf("<%s>", token)
    ;:abc::def:ghi:*
    printf("\n");
```

when run will print:

```
    <><abc><><def><ghi><>
```

`string = gsub(string, string|regexp, string)`

gsub performs text substitution using regular expressions. It takes the first parameter, matches it against the second parameter and then replaces the matched portion of the string with the third parameter. If the second parameter is a string it is converted to a regular expression as if the regexp function had been called. Gsub does the replacement multiple times to replace all occurrances of the pattern. It returns the new string formed by the replacement. If there is no match this is original string. The replacement string may contain the special sequence "\&" which is replaced by the string that matched the regular expression. Parenthesized portions of the regular expression may be matched by using \n where *n* is a decimal digit.

`string = implode(array)`

Returns a *string* formed from the concatenation of elements of *array*. Integers in the *array* will be interpreted as character codes; strings in the array will be included in the concatenation directly. Other types are ignored.

`struct = include(string [, scope])`

Parses the code contained in the file named by the string into the scope. If scope is not passed the current scope is used. Include always returns the scope into which the code was parsed. The file is opened by calling the current definition of the ICI fopen() function so path searching can be implemented by overriding that function.

`value = int(any)`

Returns an integer interpretation of *x*, or 0 if no reasonable interpretation exists. *x* should be an int, a float, or a string, else 0 will be returned.

`subpart = interval(str_or_array, start [, length])`

Returns a sub-interval of *str_or_array*, which may be either a string or an array.

If *start* (an integer) is positive the sub-interval starts at that offset (offset 0 is the first element). If *start* is negative the sub-interval starts that many elements from the end of the string (offset -1 is the last element, -2 the second last etc).

If *length* is absent, all the elements from the *start* are included in the interval. Otherwise that many elements are included (or till the end, whichever is smaller).

For example, the last character in a string can be accessed with:

```
    last = interval(str, -1);
```

And the first three elements of an array with:

```
    first3 = interval(ary, 0, 3);
```

`isatom(any)`

Return 1 (one) if *any* is an atomic (read-only) object, else 0 (zero). Note that integers, floats and strings are always atomic.

`array = keys(struct)`

Returns an array of all the keys from *struct*. The order is not predictable, but is repeatable if no elements are added or deleted from the struct between calls and is the same order as taken by a *forall* loop.

`float = log(x)`

Returns the natural logarithm of *x* (a float).

`float = log10(x)`

Returns the log base 10 of *x* (a float).

`mem = mem(start, nwords [, wordz])`

Returns a memory object which refers to a particular area of memory in the ICI interpreter's address space. Note that this is a highly dangerous operation. Many implementations will not include this function or restrict its use. It is designed for diagnostics, embedded systems and controllers. See the *alloc* function above.

`file = mopen(mem [, mode])`

Returns a *file*, which when read will fetch successive bytes from the given *memory object*. The memory object must have an access size of one (see *alloc* and *mem* above). The file is read-only and the *mode*, if passed, must be one of **`"r"`** or **`"rb"`**.

`int = nels(any)`

Returns the number of elements in *any*. The exact meaning depends on the type of *any*. If *any* is an:

array	the length of the array is returned; if it is a
struct	the number of key/value pairs is returned; if it is a
set	the number of elements is returned; if it is a
string	the number of characters is returned; and if it is a
mem	the number of words (either 1, 2 or 4 byte quantities) is returned;

and if it is anything else, one is returned.

`number = num(x)`

If *x* is an int or float, it is returned directly. If *x* is a string it will be converted to an int or float depending on its appearance; applying octal and hex interpretations according to the normal ICI source parsing conventions. (That is, if it starts with a 0x it will be interpreted as a hex number, else if it starts with a 0 it will be interpreted as an octal number, else it will be interpreted as a decimal number.)

If *x* can not be interpreted as a number the error *%s is not a number* is generated.

`scope = parse(source [, scope])`

Parses *source* in a new variable scope, or, if *scope* (a struct) is supplied, in that scope. *Source* may either be a file or a string, and in either case it is the source of text for the parse. If the parse is successful, the variables scope structure of the sub-module is returned. If an explicit scope was supplied this will be that structure.

If *scope* is not supplied a new struct is created for the auto variables. This structure in turn is given a new structure as its super struct for the static variables. Finally, this structure's super is set to the current static variables. Thus the static variables of the current module form the externs of the sub-module.

If *scope* is supplied it is used directly as the scope for the sub-module. Thus the base structure will be the struct for autos, its super will be the struct for statics etc.

For example:

```
    static x = 123;
    parse("static x = 456;", scope());
    printf("x = %d\n", x);
```

When run will print:

```
    x = 456
```

Whereas:

```
    static x = 123;
    parse("static x = 456;");
    printf("x = %d\n", x);
```

When run will print:

```
    x = 123
```

Note that while the following will work:

```
    parse(fopen("my-module.ici"));
```

It is preferable in a large program to use:

```
    parse(file = fopen("my-module.ici"));
    close(file);
```

In the first case the file will eventually be closed by garbage collection, but exactly when this will happen is unpredictable. The underlying system may only allow a limited number of simultaneous open files. Thus if the program continues to open files in this fashion a system limit may be reached before the unused files are garbage collected.

`any = pop(array)`

Returns the last element of *array* and reduces the length of *array* by one. If the array was empty to start with, NULL is returned.

`file = popen(string, [flags])`

Executes a new process, specified as a shell command line as for the *system* function, and returns a file that either reads or writes to the standard input or output of the process according to *mode*. If mode is **`“r”`** the reading from the file reads from the standard output of the process. If mode is **`“w”`** writing to the file writes to the standard input of the process. If mode is not speicified it defaults to **`“r”`**.

`float = pow(x, y)`

Returns *x^y* where both *x* and *y* are floats.

`printf([file,] fmt, args...)`

Formats a string based on *fmt* and *args* as per *sprintf* (below) and outputs the result to the *file* or to the current value of the *stdout* variable in the current scope if the first parameter is not a file. The current stdout must be a file. See *sprintf*.

`any = push(array, any)`

Appends *any* to *array*, increasing its length in the process. Returns *any*.

`put(string [, file])`

Outputs string to *file*. If *file* is not passed the current value of *stdout* in the current scope is used.

`int = rand([seed])`

Returns an pseudo random integer in the range 0..0x7FFF. If *seed* (an int) is supplied the random number generator is first seeded with that number. The sequence is predictable based on a given seed.

`reclaim()`

Force a garbage collection to occur.

`re = regexp(string [, int])`

Returns a compiled regular expression derived from *string* This is the method of generating regular expressions at run-time, as opposed to the direct lexical form. For example, the following three expressions are similar:

```
    str ~ #*\.c#
    str ~ regexp("*\\.c");
    str ~ $regexp("*\\.c");
```

except that the middle form computes the regular expression each time it is executed. Note that when a regular expression includes a # character the *regexp* function must be used, as the direct lexical form has no method of escaping a #.

The optional second parameter is a bit-set that controls various aspects of the compiled regular expression's behaviour. This value is passed directly to the PCRE package's regular expression compilation function. Presently no symbolic names are defined for the possible values and interested parties are directed to the PCRE documention included with the ICI source code.

Note that regular expressions are intrinsically atomic. Also note that non-equal strings may sometimes compile to the same regular expression.

`re = regexpi(string [, int])`

Returns a compiled regular expression derived from *string* that is case-insensitive. I.e., the regexp will match a string regardless of the case of alphabetic characters. Literal regular expressions to perform case-insensitive matching may be constructed using the special PCRE notation for such purposes, see page 75.

`remove(string)`

Deletes the file whose name is given in *string*.

`current = scope([replacement])`

Returns the current scope structure. This is a struct whose base element holds the auto variables, the super of that hold the statics, the super of that holds the externs etc. Note that this is a real reference to the current scope structure. Changing, adding and deleting elements of these structures will affect the values and presence of variables in the current scope.

If a *replacement* is given, that struct replaces the current scope structure, with the obvious implications. This should clearly be used with caution. Replacing the current scope with a structure which has no reference to the standard functions also has the obvious effect.

`int = seek(file, int, int)`

Set the input/output position for a file and returns the new I/O position or -1 if an error ocurred. The arguments are the same as for the C library's **fseek** function. If the file object does not support setting the I/O position or the seek operation fails an error is raised.

`set = set(any...)`

Returns a set formed from all the arguments. For example:

```
    set()
```

will return a new empty set; and

```
set(1, 2, "a string")
```

will return a new set with three elements, *1*, *2*, and *"the string"*.

This is the run-time equivalent of the set literal. Thus the following two expressions are equivalent:

```
$set(1, 2, "a string")

[set 1, 2, "a string"]
```

`x = sin(angle)`

Returns the sine of *angle* (a float interpreted in radians).

`file = sopen(string [, mode])`

Returns a *file*, which when read will fetch successive characters from the given *string*. The file is read-only and the *mode*, if passed, must be one of **`“r”`** or **`“rb”`**.

Files are, in general, system dependent. This is the only standard routine which opens a file. But on systems that support byte stream files, the function *fopen* will be set to the most appropriate method of opening a file for general use. The interpretation of *mode* is largely system dependent, but the strings *"r"*, *"w"*, and *"rw"* should be used for read, write, and read-write file access respectively.

`sort(array, func)`

Sort the content of the array using the heap sort algorithm with func as the comparison function. The comparison function is called with two elements of the array as parameters, *a* and *b*. If *a* is equal to *b* the function should return zero. If *a* is less than *b*, -1, and if *a* is greater than *b*, 1.

For example,

```
static cmp(a, b)
{
    if (a == b)
        return 0;
    if (a < b)
        return -1;
    return 1;
}

static a = array(1, 3, 2);

sort(a, cmp);
```

`string = sprintf(fmt, args...)`

Return a formatted string based on *fmt* (a string) and *args*. Most of the usual % format escapes of ANSI C printf are supported. In particular; the integer format letters *diouxXc* are supported, but if a float is provided it will be converted to an int. The floating point format letters *feEgG* are supported, but if the argument is an int it will be converted to a float. The string format letter, *s* is supported and requires a string. Finally the *%* format to get a single *%* works.

The flags, precision, and field width options are supported. The indirect field width and precision options with * also work and the corresponding argument must be an int.

For example:

```
    sprintf("%08X <%4s> <%-4s>", 123, "ab", "cd")
```

will produce the string:

```
    0000007B <  ab> <cd  >
```

and

```
    sprintf("%0*X", 4, 123)
```

will produce the string:

```
    007B
```

`x = sqrt(float)`

Returns the square root of *float*.

`string = string(any)`

Returns a short textual representation of *any*. If *any* is an int or float it is converted as if by a *%d* or *%g* format. If it is a string it is returned directly. Any other type will returns its type name surrounded by angle brackets, as in *<struct>*.

`struct = struct([super,] key, value...)`

Returns a new structure. This is the run-time equivalent of the struct literal. If there are an odd number of arguments the first is used as the super of the new struct; it must be a struct. The remaining pairs of arguments are treated as key and value pairs to initialise the structure with; they may be of any type. For example:

```
    struct()
```

returns a new empty struct;

```
    struct(anotherStruct)
```

returns a new empty struct which has *anotherStruct* as its super;

```
    struct("a", 1, "b", 2)
```

returns a new struct which has two entries *a* and *b* with the values *1* and *2*; and

```
    struct(anotherStruct, "a", 1, "b", 2)
```

returns a new struct which has two entries *a* and *b* with the values *1* and *2* and a super of *anotherStruct*.

Note that the super of the new struct is set *after* the assignments of the new elements have been made. Thus the initial elements given as arguments will not affect values in any super struct.

The following two expressions are equivalent:

```
    $struct(anotherStruct, "a", 1, "b", 2)

    [struct:anotherStruct, a = 1, b = 2]
```

`string = sub(string, string|regexp, string)`

Sub performs text substitution using regular expressions. It takes the first parameter, matches it against the second parameter and then replaces the matched portion of the string with the third parameter. If the second parameter is a string it is converted to a regular expression as if the regexp function had been called. Sub does the replacement once (unlike gsub). It returns the new string formed by the replacement. If there is no match this is original string. The replacement string may contain the special sequence "\&" which is replaced by the string that matched the regular expression. Parenthesized portions of the regular expression may be matched by using \n where *n* is a decimal digit.

`current = super(struct [, replacement])`

Returns the current super struct of *struct*, and, if *replacement* is supplied, sets it to a new value. If *replacement* is NULL any current super struct reference is cleared (that is, after this *struct* will have no super).

`x = tan(angle)`

Returns the tangent of *angle* (a float interpreted in radians).

`foat = now()`

Returns the current time expressed as a signed float time in seconds since 0:00, 1st Jan 2000 UTC.

`float|struct = calendar(struct|float)`

Converts between calendar time and arithmetic time. An arithmetic time is expressed as a signed float time in seconds since 0:00, 1st Jan 2000 UTC. The calendar time is expressed as a structure with fields revealing the local (including current daylight saving adjustment) calendar date and time. Fields in the calendar structure are:

second	The float number of seconds after the minute.
minute	The int number of minutes after the hour.
hour	The int number of hours since midnight.

day	The day of the month (1..31).
month	The int month number, Jan is 0.
year	The int year.
wday	The day since Sunday (0..6)
yday	Days since 1st Jan.

When converting from a local calendar time to an arithmetic time, the fields *sec*, *min*, *hour*, *mday*, *mon*, *year* are used. They need not be restricted to their nomal ranges.

`string = tochar(int)`

Returns a one character string made from the character code specified by *int*.

`int = toint(string)`

Returns the character code of the first character of *string*.

`string = typeof(any)`

Returns the type name (a string) of *any*. See the section on types above for the possible type names.

`array = vstack()`

Returns a representation of the call stack of the current program at the time of the call. It can be used to perform stack tracebacks and related debugging operations. The result is an array of structures, each of which is a variable scope (see *scope*) structure of succesively deeper nestings of the current function nesting.

`event = waitfor(event...)`

Blocks (waits) until an *event* indicated by any of its arguments occurs, then returns that argument. The interpretation of an event depends on the nature of each argument. A file argument is triggered when input is available on the file. A float argument waits for that many seconds to expire, an int for that many millisecond (they then return 0, not the argument given). Other interpretations are implementation dependent. Where several events occur simultaneously, the first as listed in the arguments will be returned.

Note that in some implementations some file types may always appear ready for input, despite the fact that they are not.

Command Line Arguments

Versions of ICI on systems that support passing parameters from the command line provide two predefined variables, argv and argc, for accessing these arguments.

On Win32 platforms ICI performs wildcard expansion in the traditional MS-DOS fashion. Arguments containing wildcard meta-characters, ‘?’ and ‘*’, may be protected by enclosing them in single or double quotes.

argv

An array of strings containing the command line arguments. The first element is the name of the ICI program and subsequent elements are the arguments passed to that program.

argc

The count of the number of elements in argv. Initially equal to nels(argv).

Unix System Calls

Most Unix implementation of ICI provide access to many of the Unix system calls and other useful C library functions. Note that not all system calls are supported and those that are may be incompletely supported (e.g., *signal*). Most system call functions return integers, zero if the call succeeded. Errors are reported using ICI's error handling and "system calls" will never return the -1 error return value. If an error is raised by a system call the value of "error" in the error handler will be the error message (as printed by the perror(3) function or returned by the ANSI C strerror() function).

To assist in the use of system calls ICI pre-defines variables to hold the various flags and other values used when calling the system calls. These variables are equivalent to the macros used in C. Not all systems support all these variables. If the C header files do not define a value then ICI will not pre-define the variable.

Win32 Support

The version of ICI for Microsoft's 32-bit Windows platforms (Win32) supports many of these functions. Functions supported on Win32 platforms (Windows 95 and Windows NT) are marked with **WIN32**. In addition some functions are only available on Win32 platforms and are marked as so.

The following list summarises the Unix system call interface pre-defined variables. See the documentation for the C macros for information as to their use.

Values for open's *flags* parameter,

O_RDONLY	
O_WRONLY	
O_RDWR	
O_APPEND	
O_CREAT	
O_TRUNC	
O_EXCL	
O_SYNC	
O_NDELAY	
O_NONBLOCK	
O_BINARY	(WIN32 only)

Values for spawn's *mode* parameter,

_P_WAIT	(WIN32 only)
_P_NOWAIT	(WIN32 only)

Values for access's *mode* parameter,

R_OK
W_OK
X_OK
F_OK

Values for lseek's *whence* parameter,

SEEK_SET
SEEK_CUR
SEEK_END

The following list summarises the system interface functions. Following this is a detailed descriptions of each of them.

int =	access(string *[*, int*]*)
int =	creat(string, int)
array =	dir([string,] [string,] [regexp])
int =	dup(int *[*, int*]*)
	exec(string, array)
	exec(string, string...)
int =	lseek(int, int *[*, int*]*)
int =	open(string, int *[*, int*]*)
array =	pipe()
struct =	stat(string|int|file)
int =	wait()
string =	ctime(int)
int =	time()
file =	fdopen(int)
string =	getcwd()
	alarm(int)
	acct(string)
	chdir(string)
	chmod(string, int)
	chown(string, int, int)
	chroot(string)
	_close(int)

	_exit(int)
int =	fork()
int =	getpid()
int =	getpgrp()
int =	getppid()
int =	getuid()
int =	geteuid()
int =	getgid()
int =	getegid()
	kill(int, int)
	link(string, string)
	mkdir(string, int)
	mknod(string, int, int)
	nice(int)
	pause()
	rmdir(string)
	setpgrp()
	setuid(int)
	setgid(int)
	signal(int, int)
	sync()
	ulimit(int, int)
	umask(int)
	unlink(string)
	clock()
	system(string)
	lockf(int, int, int)
	sleep(int)
int =	spawn([int,] string, string...)
int =	spawn([int,] string, array)
	rename(string, string)
struct =	passwd(int|string)
array =	passwd()

int = access(string *[*, int*]*)

Call the access(2) function to determine the accessibility of a file. The first parameter is the pathname of the file system object to be tested. The second, optional, parameter is the *mode* (a bitwise combination of R_OK, W_OK and X_OK or the special value, F_OK). If *mode* is not passed F_OK is assumed. Access returns 0 if the file system object is accessible. Also supported on WIN32 platforms.

int = creat(string, int)

Create a new ordinary file with the given pathname and mode (permissions etc...) and return the file descriptor, open for writing, for the file. Also supported on WIN32 platforms.

`array = dir([string,] [string,] [regexp])`

The dir() function is used to read the contents of directories. It returns an array of strings being the names found in the directory. The first string parameter names a directory to read and defaults to “.” — the current directory. The second string parameter controls which names are returned. It may be one of “f” — return only the names of files, “d” — return the names of sub-directories, or “a” — return the names of all objects in the directory. The regexp parameter, if passed, is used to filter the returned names. Only names that match the regexp are returned. Note that when using dir() to traverse directory hierarchies that the “.” and “..” names are returned when listing the names of sub-directories, these will need to be avoided when traversing.

`int = dup(int [, int])`

Duplicate a file descriptor by calling dup(2) or dup2(2) and return a new descriptor. If only a single parameter is passed dup(2) is called otherwise dup2(2) is called. Also supported on WIN32 platforms.

`exec(string, array)`

`exec(string, string...)`

Execute a new program in the current process. The first parameter to exec is the pathname of an executable file (the program). The remaining parameters are either; an array of strings defining the parameters to be passed to the program, or, a variable number of strings that are passed, in order, to the program as its parameters. The first form is similar to C’s execv function and the second form to C’s execl functions. Note that no searching of the user’s path is performed and the environment passed to the program is that of the current process (i.e., both are implemented by calls to execv(2)). This function is available on Win32 platforms

`int = lseek(int, int [, int])`

Set the read/write position for an open file. The first parameter is the file descriptor associated with the file system object, the second parameter the offset. The third is the *whence* value which determines how the new file position is calculated. The whence value may be one of SEEK_SET, SEEK_CUR or SEEK_END and defaults to SEEK_SET if not specified. Also supported on WIN32 platforms.

`int = open(string, int [, int])`

Open the named file for reading or writing depending upon the value of the second parameter, *flags*, and return a file descriptor. The second parameter is a bitwise combination of the various O_ values (see above) and if this set includes the O_CREAT flag a third parameter, *mode*, **must** also be supplied. Also supported on WIN32 platforms.

`array = pipe()`

Create a pipe and return an array containing two, integer, file descriptors used to refer to the input and output endpoints of the pipe.

`struct = stat(string|int|file)`

Obtain information on the named file system object, file descriptor or file underlying an ICI file object and return a struct containing that information. If the parameter is a file object that file object must refer to a file opened with ICI's **`fopen`** function. The returned struct contains the following keys (which have the same names as the fields of the Unix statbuf structure with the leading "st_" prefix removed),

```
dev
ino
mode
nlink
uid
gid
rdev
size
atime
mtime
ctime
blksize
blocks
```

All values are integers. Also supported on WIN32 platforms.

`int = wait()`

Wait until a signal is received or a child process terminates or stops due to tracing and return the status returned by system call.

`string = ctime(int)`

Convert a time value (see time, below) to a string of the form "Sun Sep 16 01:03:52 1973\n" and return that string. This is primarily of use when converting the time values returned by stat. Also supported on WIN32 platforms.

`int = time()`

Return the time since 00:00:00 GMT, Jan. 1, 1970, measured in seconds. Also supported on WIN32 platforms.

`file = fdopen(int` *[*, **`mode`***]***`)`**

Returns a file object that can be used to perform I/O on the specified file descriptor. The file is opened for reading or writing according to *mode* (see *fopen*). If mode is specified ***"r"*** (reading) is assumed.

`string = getcwd()`

Returns the name of the current working directory. Also supported on WIN32 platforms.

`alarm(int)`

Schedule a SIGALRM signal to be posted to the current process in the specified number of seconds. If the parameter is zero any alarm is cancelled.

`acct(string)`

Enable accounting on the specified file.

`chdir(string)`

Change the process's current working directory to the specified path. Also supported on WIN32 platforms.

`chmod(string, int)`

Change the mode of a file system object.

`chown(string, int, int)`

Change the owner and group identifiers for a file system object.

`chroot(string)`

Change root directory for process.

`_close(int)`

Close a file descriptor. Also supported on WIN32 platforms.

`_exit(int)`

Exit the current process returning an integer exit status to the parent. Also supported on WIN32 platforms.

`int = fork()`

Create a new process. In the parent this returns the process identifier for the newly created process. In the newly created process it returns zero.

`int = getpid()`

Get the process identifier for the current process.

`int = getpgrp()`

Get the current process group identifier.

`int = getppid()`

Get the parent process identifier.

`int = getuid()`

Get the real user identifier of the owner of the current process.

`int = geteuid()`

Get the effective user identifier for the owner of the current process.

`int = getgid()`

Get the real group identifier for the current process.

`int = getegid()`

Get the effective group identifier for the current process.

`kill(int, int)`

Post a signal to a process.

`link(string, string)`

Create a link to an existing file.

`mkdir(string, int)`

Create a directory with the specified mode. Also supported on WIN32 platforms.

`mknod(string, int, int)`

Create a special file.

`nice(int)`

Change the *nice* value of a process.

`pause()`

Wait until a signal is delivered to the process.

`rmdir(string)`

Remove a directory. Also supported on WIN32 platforms.

`setpgrp()`

Set the process group.

`setuid(int)`

Set the real and effective user identifier for the current process.

`setgid(int)`

Set the real and effective group identifier for the current process.

`signal(int, int)`

Control signal handling in the process. Note at present handlers cannot be installed so signals are of limited use in ICI programs.

`sync()`

Schedule in-memory file data to be written to disk.

`ulimit(int, int)`

Get and set user limits.

`umask(int)`

Set file creation mask.

`unlink(string)`

Remove a file. Also supported on WIN32 platforms.

`system(string)`

Execute a system command and return its exit status. Also supported on WIN32 platforms however using the system's command interpreter.

`sleep(int)`

Suspend the process for the specified number of seconds.

int = spawn([mode,] string, string...)

int = spawn([mode,] string, array)

int = spawnp([mode,] string, string...)

int = spawnp([mode,] string, array)

Spawn a sub-process. The parameters, other than mode, are as for exec - the string is the name of the executable and the remaining parameters form the command line arguments passed to the executable.

The mode parameter controls whether or not the parent process waits for the spawned process to termiante. If mode is _P_WAIT the call to spawn returns when the process terminates and the result of spawn is the process exit status. If mode is not passed or is _P_NOWAIT the call to spawn returns prior to the process terminating and the result is the Win32 process handle for the new process.

The *spawnp* variant will search the directories listed in the PATH environment variable for the executable program. In all other respects it is indentical to spawn.

This function is only available on Win32 platforms.

rename(string, string)

Change the name of a file. The first parameter is the name of an existing file and the second is the new name that it is to be given.

struct = passwd(int | string)

array = passwd()

The passwd() function accesses the Unix password file (which may or may not be an actual file according to the local system configuration). With no parameters passwd() returns an array of all password file entries, each entry is a struct. With a parameter passwd() returns the entry for the specific user id., int parameter, or user name, string parameter. A password file entry is a struct with the following keys and values,

```
    name        The user's login name, a string.
    passwd      The user's encrypted password, a string.
```

```
               Note that some systems protect this (shadow
               password files) and this field may not be an
               actual encrypted password.
uid            The user id., an int.
gid            The user’s default group, an int.
gecos          The so-called gecos field, a string.
dir            The user’s home directory, a string.
shell          The user’s shell (initial program), a
               string.
```

Sockets Interface

The *sockets* extension is available on systems that provide BSD-compatible sockets calls and for Win32 platforms. The extension allows ICI programs to access network functions. The sockets extension is generally compatible with the C sockets functions but uses types and calling semantics more akin to the ICI environment.

The sockets extension introduces a new type, *socket*, to hold socket objects. The new intrinsic function, *socket*, returns a socket object.

Network Addresses

The sockets interfaces specifies IP network addresses using strings. Network addresses are of the form *port@ host* where the @host part is optional. The port may be specified as an integer number or a string which is looked up in the services database. If the port is a service name it may be in the form *name*/*protocol* with protocol being either *tcp* or *udp*. The host portion of the address may be a domain name, an IP address in dotted decimal notation or one of the special addresses local (“.” - dot), any (“?”) or all (“*”). If the host portion is omitted the default host depends on the context. See the descriptions of the *connect* and *bind* functions below.

The following list summarises the sockets interface functions. Following this is a detailed descriptions of each of them.

skt =	socket(string)
skt =	listen(skt)
skt =	accept(skt)
skt =	connect(skt, string)
skt =	bind(skt, string)
struct =	select([int,] set [, set [, set]])
int =	getsockopt(skt, string)
	setsockopt(skt, string, int)
string =	domainname()
string =	hostname()
string =	username([int])
string =	getpeername(skt)
string =	getsockname(skt)
	sendto(skt, string, string)
struct =	recvfrom(skt, int)

	send(skt, string)
string =	recv(skt, int)
int =	getportno(skt)
string =	gethostbyname(string)
int =	sktno(skt)
file =	sktopen(skt [, mode])
array =	socketpair()

skt = socket(string)

Create and return a new socket object of the specified protocol. The string, the protocol, may be one of *tcp* or *udp*. For example,

```
    skt = socket(“tcp”);
```

skt = accept(skt)

Accept a connection to a TCP socket and return a new socket for that connection.

skt = listen(skt)

Allow connections to a TCP socket. Returns the socket passed.

skt = connect(skt, address)

Establish a TCP connection to the specified address or associate the address with as the destination for messages on a UDP socket. If the host portion of the address is not specified “.” (dot) is used to connect to the local host. The original socket is returned.

skt = bind(skt [, address|int])

Associate a local address for the socket (TCP or UDP). If the address is not specified the system selects an unused local port number for the socket. If the host portion of the address is not specified “?” (any) is used. If the address is passed as an integer it specifies the port number to be bound, the host portion is “?”. Bind returns the socket parameter.

struct = select([int,] set|NULL [, set|NULL [, set|NULL]])

Check sockets for I/O readiness with optional timeout. Select may be passed up to three sets of sockets that are checked for readiness to perform I/O. The first set holds the sockets to test for input pending, the second set the sockets to test for output able and the third set the sockets to test for exceptional states. NULL may be passed in place of a set parameter to avoid passing empty sets. An integer may also appear in the parameter list. This integer specifies the number of milliseconds to wait for the sockets to become ready. If a zero timeout is passed the sockets are polled to test their state. If no timeout is passed the call blocks until at least one of the sockets is ready for I/O.

The result of select is a struct containing three sets, of sockets, identified by the keys `read`, `write` and `except`.

int = getsockopt(skt, string, int)

Retrieve the value of a socket *option*. A socket may have various attributes associated with it. These are accessed via the getsockopt and setsockopt functions. The attributes are identified using string keys from the following list,

```
    debug
    reuseaddr
    keepalive
    dontroute
    useloopback
    linger
    broadcast
    oobinline
    sndbuf
    rcvbuf
    type
    error
```

setsockopt(skt, string, int)

Set a socket option (see getsockopt for option names) to the integer value.

string = domainname()

Return the domain name of the current host.

string = hostname()

Return the name of the current host.

string = username([int])

Return the name of the owner of the current process or if an integer, user number, is passed, of that user.

string = getpeername(skt)

Return the address of the *peer* of a TCP socket.

string = getsockname(skt)

Return the local address of a socket.

sendto(skt, string, string)

Send the data in the second parameter to the specified address.

array = socketpair()

Returns an array containing a pair of connected sockets.

struct = recvfrom(skt, int)

Receive a message on a socket and return a struct containing the data of the message, in string, and the source address of the data. The int parameter gives the maximum number of bytes to receive. The result is a struct with the keys `msg` and `addr` used to access the returned information.

send(skt, string)

Send the content of the string on a socket.

string = recv(skt, int)

Receive data from a socket and return it as a string. The int parameter fives the maximum size of message that will be received.

int = getportno(skt)

Return the local port number assigned to a TCP or UDP socket.

string = gethostbyname(string)

Match a network address against the hosts database and return a hostname.

int = sktno(skt)

Return the file descriptor associated with a socket.

file = sktopen(skt [, mode])

Open a socket as a file, for input or output according to *mode* (see *fopen*). This function is **not** available on WIN32 platforms.

Regular Expression Syntax

ICI uses Philip Hazel's PCRE (Perl-compatible regular expressions) package. The following is extracted from the file `pcre.3.txt` included with the PCRE distribution. This document is intended to be used with the PCRE C functions and makes reference to a number of constants that may be used as option specifiers to the C functions (all such constants are prefixed with the string "`PCRE_`"). These constants are not available in the ICI interface at time of writing although the `regexp()` function does allow a numeric option specific to be passed.

> The syntax and semantics of the regular expressions supported by PCRE are described below. Regular expressions are also described in the Perl documentation and in a number of other books, some of which have copious examples. Jeffrey Friedl's "Mastering Regular Expressions", published by O'Reilly (ISBN 1-56592-257-3), covers them in great detail. The description here is intended as reference documentation.
>
> A regular expression is a pattern that is matched against a subject string from left to right. Most characters stand for themselves in a pattern, and match the corresponding characters in the subject. As a trivial example, the pattern
>
> The quick brown fox
>
> matches a portion of a subject string that is identical to itself. The power of regular expressions comes from the ability to include alternatives and repetitions in the pattern. These are encoded in the pattern by the use of meta-characters, which do not stand for themselves but instead are interpreted in some special way.
>
> There are two different sets of meta-characters: those that are recognized anywhere in the pattern except within square brackets, and those that are recognized in square brackets. Outside square brackets, the meta-characters are as follows:

\	general escape character with several uses
^	assert start of subject (or line, in multiline mode)
$	assert end of subject (or line, in multiline mode)
.	match any character except newline (by default)
[	start character class definition
|	start of alternative branch
(	start subpattern
)	end subpattern
?	extends the meaning of (
	also 0 or 1 quantifier
	also quantifier minimizer
*	0 or more quantifier
+	1 or more quantifier
{	start min/max quantifier

Part of a pattern that is in square brackets is called a “character class”. In a character class the only meta-characters are:

\	general escape character
^	negate the class, but only if the first character
-	indicates character range
]	terminates the character class

The following sections describe the use of each of the meta-characters.

BACKSLASH

The backslash character has several uses. Firstly, if it is followed by a non-alphameric character, it takes away any special meaning that character may have. This use of backslash as an escape character applies both inside and outside character classes.

For example, if you want to match a “*” character, you write “*” in the pattern. This applies whether or not the following character would otherwise be interpreted as a meta-character, so it is always safe to precede a non-alphameric with “\” to specify that it stands for itself. In particular, if you want to match a backslash, you write “\\”.

If a pattern is compiled with the PCRE_EXTENDED option, whitespace in the pattern (other than in a character class) and characters between a “#” outside a character class and the next newline character are ignored. An escaping backslash can be used to include a whitespace or “#” character as part of the pattern.

A second use of backslash provides a way of encoding non-printing characters in patterns in a visible manner. There is no restriction on the appearance of non-printing characters, apart from the binary zero that terminates a pattern, but when a pattern is being prepared by text editing, it is usually easier to use one of the following escape sequences than the binary character it represents:

\a	alarm, that is, the BEL character (hex 07)
\cx	“control-x”, where x is any character
\e	escape (hex 1B)
\f	formfeed (hex 0C)
\n	newline (hex 0A)
\r	carriage return (hex 0D)
\t	tab (hex 09)
\xhh	character with hex code hh
\ddd	character with octal code ddd, or backreference

The precise effect of “\cx” is as follows: if “x” is a lower case letter, it is converted to upper case. Then bit 6 of the character (hex 40) is inverted. Thus “\cz” becomes hex 1A, but “\c{“ becomes hex 3B, while “\c;” becomes hex 7B.

After “\x”, up to two hexadecimal digits are read (letters can be in upper or lower case).

After “\0” up to two further octal digits are read. In both cases, if there are fewer than two digits, just those that are present are used. Thus the sequence “\0\x\07” specifies two binary zeros followed by a BEL character. Make sure you supply two digits after the initial zero if the character that follows is itself an octal digit.

The handling of a backslash followed by a digit other than 0 is complicated. Outside a character class, PCRE reads it and any following digits as a decimal number. If the number is less than 10, or if there have been at least that many previous capturing left parentheses in the expression, the entire sequence is taken as a back reference. A description of how this works is given later, following the discussion of parenthesized subpatterns.

Inside a character class, or if the decimal number is greater than 9 and there have not been that many capturing subpatterns, PCRE re-reads up to three octal digits following the backslash, and generates a single byte from the least significant 8 bits of the value. Any subsequent digits stand for themselves. For example:

\040	is another way of writing a space
\40	is the same, provided there are fewer than 40 previous capturing subpatterns
\7	is always a back reference
\11	might be a back reference, or another way of writing a tab

\011   is always a tab

\0113 is a tab followed by the character “3”

\113   is the character with octal code 113 (since there can be no more than 99 back references)

\377   is a byte consisting entirely of 1 bits

\81   is either a back reference, or a binary zero followed by the two characters “8” and “1”

Note that octal values of 100 or greater must not be introduced by a leading zero, because no more than three octal digits are ever read.

All the sequences that define a single byte value can be used both inside and outside character classes. In addition, inside a character class, the sequence “\b” is interpreted as the backspace character (hex 08). Outside a character class it has a different meaning (see below).

The third use of backslash is for specifying generic character types:

\d   any decimal digit

\D   any character that is not a decimal digit any whitespace character

\S   any character that is not a whitespace character

\w   any “word” character

\W   any “non-word” character

Each pair of escape sequences partitions the complete set of characters into two disjoint sets. Any given character matches one, and only one, of each pair.

A “word” character is any letter or digit or the underscore character, that is, any character which can be part of a Perl “word”. The definition of letters and digits is controlled by PCRE’s character tables, and may vary if locale-specific matching is taking place (see “Locale support” above). For example, in the “fr” (French) locale, some character codes greater than 128 are used for accented letters, and these are matched by \w.

These character type sequences can appear both inside and outside character classes. They each match one character of the appropriate type. If the current matching point is at the end of the subject string, all of them fail, since there is no character to match.

The fourth use of backslash is for certain simple assertions. An assertion specifies a condition that has to be met at a particular point in a match, without consuming any characters from the subject string. The use of subpatterns for more complicated assertions is described below. The backslashed assertions are

\b   word boundary

\B   not a word boundary

\A   start of subject (independent of multiline mode)

\Z end of subject or newline at end (independent of multiline mode)

\z end of subject (independent of multiline mode)

These assertions may not appear in character classes (but note that “\b” has a different meaning, namely the backspace character, inside a character class).

A word boundary is a position in the subject string where the current character and the previous character do not both match \w or \W (i.e. one matches \w and the other matches \W), or the start or end of the string if the first or last character matches \w, respectively.

The \A, \Z, and \z assertions differ from the traditional circumflex and dollar (described below) in that they only ever match at the very start and end of the subject string, whatever options are set. They are not affected by the PCRE_NOTBOL or PCRE_NOTEOL options. If the startoffset argument of pcre_exec() is non-zero, \A can never match. The difference between \Z and \z is that \Z matches before a newline that is the last character of the string as well as at the end of the string, whereas \z matches only at the end.

CIRCUMFLEX AND DOLLAR

Outside a character class, in the default matching mode, the circumflex character is an assertion which is true only if the current matching point is at the start of the subject string. If the startoffset argument of pcre_exec() is non-zero, circumflex can never match. Inside a character class, circumflex has an entirely different meaning (see below).

Circumflex need not be the first character of the pattern if a number of alternatives are involved, but it should be the first thing in each alternative in which it appears if the pattern is ever to match that branch. If all possible alternatives start with a circumflex, that is, if the pattern is constrained to match only at the start of the subject, it is said to be an “anchored” pattern. (There are also other constructs that can cause a pattern to be anchored.)

A dollar character is an assertion which is true only if the current matching point is at the end of the subject string, or immediately before a newline character that is the last character in the string (by default). Dollar need not be the last character of the pattern if a number of alternatives are involved, but it should be the last item in any branch in which it appears. Dollar has no special meaning in a character class.

The meaning of dollar can be changed so that it matches only at the very end of the string, by setting the PCRE_DOLLAR_ENDONLY option at compile or matching time. This does not affect the \Z assertion.

The meanings of the circumflex and dollar characters are changed if the PCRE_MULTILINE option is set. When this is the case, they match immediately after and immediately before an internal “\n” character, respectively, in addition to matching at the start and end of the subject string. For example, the pattern /^abc$/ matches the subject string “def\nabc” in multiline mode, but not otherwise. Consequently, patterns that are anchored in single line mode because all branches start with “^” are not anchored in multiline mode, and a match for circumflex is possible when the startoffset argument of pcre_exec() is non-zero. The PCRE_DOLLAR_ENDONLY option is ignored if PCRE_MULTILINE is set.

Note that the sequences \A, \Z, and \z can be used to match the start and end of the subject in both modes, and if all branches of a pattern start with \A is it always anchored, whether

PCRE_MULTILINE is set or not.

FULL STOP (PERIOD, DOT)

Outside a character class, a dot in the pattern matches any one character in the subject, including a non-printing character, but not (by default) newline. If the PCRE_DOTALL option is set, then dots match newlines as well. The handling of dot is entirely independent of the handling of circumflex and dollar, the only relationship being that they both involve newline characters. Dot has no special meaning in a character class.

SQUARE BRACKETS

An opening square bracket introduces a character class, terminated by a closing square bracket. A closing square bracket on its own is not special. If a closing square bracket is required as a member of the class, it should be the first data character in the class (after an initial circumflex, if present) or escaped with a backslash.

A character class matches a single character in the subject; the character must be in the set of characters defined by the class, unless the first character in the class is a circumflex, in which case the subject character must not be in the set defined by the class. If a circumflex is actually required as a member of the class, ensure it is not the first character, or escape it with a backslash.

For example, the character class [aeiou] matches any lower case vowel, while [^aeiou] matches any character that is not a lower case vowel. Note that a circumflex is just a convenient notation for specifying the characters which are in the class by enumerating those that are not. It is not an assertion: it still consumes a character from the subject string, and fails if the current pointer is at the end of the string.

When caseless matching is set, any letters in a class represent both their upper case and lower case versions, so for example, a caseless [aeiou] matches “A” as well as “a”, and a caseless [^aeiou] does not match “A”, whereas a caseful version would.

The newline character is never treated in any special way in character classes, whatever the setting of the PCRE_DOTALL or PCRE_MULTILINE options is. A class such as [^a] will always match a newline.

The minus (hyphen) character can be used to specify a range of characters in a character class. For example, [d-m] matches any letter between d and m, inclusive. If a minus character is required in a class, it must be escaped with a backslash or appear in a position where it cannot be interpreted as indicating a range, typically as the first or last character in the class.

It is not possible to have the literal character “]” as the end character of a range. A pattern such as [W-]46] is interpreted as a class of two characters (“W” and “-”) followed by a literal string “46]”, so it would match “W46]” or “-46]”. However, if the “]” is escaped with a backslash it is interpreted as the end of range, so [W-\]46] is interpreted as a single class containing a range followed by two separate characters. The octal or hexadecimal representation of “]” can also be used to end a range.

Ranges operate in ASCII collating sequence. They can also be used for characters specified numerically, for example [\000-\037]. If a range that includes letters is used when caseless matching is set, it matches the letters in either case. For example, [W-c] is equivalent to [][\^_`wxyzabc], matched caselessly, and if character tables for the “fr” locale are in use, [\xc8-\xcb] matches accented E characters in both cases.

The character types \d, \D, \s, \S, \w, and \W may also appear in a character class, and add the characters that they match to the class. For example, [\dABCDEF] matches any hexadecimal digit. A circumflex can conveniently be used with the upper case character types to specify a more restricted set of characters than the matching lower case type. For example, the class [^\W_] matches any letter or digit, but not underscore.

All non-alphameric characters other than \, -, ^ (at the start) and the terminating] are non-special in character classes, but it does no harm if they are escaped.

VERTICAL BAR

Vertical bar characters are used to separate alternative patterns. For example, the pattern

```
gilbert|sullivan
```

matches either “gilbert” or “sullivan”. Any number of alternatives may appear, and an empty alternative is permitted (matching the empty string). The matching process tries each alternative in turn, from left to right, and the first one that succeeds is used. If the alternatives are within a subpattern (defined below), “succeeds” means matching the rest of the main pattern as well as the alternative in the subpattern.

INTERNAL OPTION SETTING

The settings of PCRE_CASELESS, PCRE_MULTILINE, PCRE_DOTALL, and PCRE_EXTENDED can be changed from within the pattern by a sequence of Perl option letters enclosed between “(?” and “)”. The option letters are

```
i  for PCRE_CASELESS

m  for PCRE_MULTILINE

s  for PCRE_DOTALL

x  for PCRE_EXTENDED
```

For example, (?im) sets caseless, multiline matching. It is also possible to unset these options by preceding the letter with a hyphen, and a combined setting and unsetting such as (?im-sx), which sets PCRE_CASELESS and PCRE_MULTILINE while unsetting PCRE_DOTALL and PCRE_EXTENDED, is also permitted. If a letter appears both before and after the hyphen, the option is unset.

The scope of these option changes depends on where in the pattern the setting occurs. For settings that are outside any subpattern (defined below), the effect is the same as if the options were set or unset at the start of matching. The following patterns all behave in exactly the same way:

```
  (?i)abc   a(?i)bc   ab(?i)c   abc(?i)
```

which in turn is the same as compiling the pattern abc with PCRE_CASELESS set. In other words, such “top level” settings apply to the whole pattern (unless there are other changes inside subpatterns). If there is more than one setting of the same option at top level, the rightmost setting is used.

If an option change occurs inside a subpattern, the effect is different. This is a change of behaviour in Perl 5.005. An option change inside a subpattern affects only that part of the subpattern that follows it, so

```
  (a(?i)b)c
```

matches abc and aBc and no other strings (assuming PCRE_CASELESS is not used). By this means, options can be made to have different settings in different parts of the pattern. Any changes made in one alternative do carry on into subsequent branches within the same subpattern. For example,

```
  (a(?i)b|c)
```

matches “ab”, “aB”, “c”, and “C”, even though when matching “C” the first branch is abandoned before the option setting. This is because the effects of option settings happen at compile time. There would be some very weird behaviour otherwise.

The PCRE-specific options PCRE_UNGREEDY and PCRE_EXTRA can be changed in the same way as the Perl-compatible options by using the characters U and X respectively. The (?X) flag setting is special in that it must always occur earlier in the pattern than any of the additional features it turns on, even when it is at top level. It is best put at the start.

SUBPATTERNS

Subpatterns are delimited by parentheses (round brackets), which can be nested. Marking part of a pattern as a subpattern does two things:

1. It localizes a set of alternatives. For example, the pattern

```
  cat(aract|erpillar|)
```

matches one of the words “cat”, “cataract”, or “caterpillar”. Without the parentheses, it would match “cataract”, “erpillar” or the empty string.

2. It sets up the subpattern as a capturing subpattern (as defined above). When the whole pattern matches, that portion of the subject string that matched the subpattern is passed back to the caller via the ovector argument of pcre_exec(). Opening parentheses are counted from left to right (starting from 1) to obtain the numbers of the capturing subpatterns.

For example, if the string “the red king” is matched against the pattern

```
  the ((red|white) (king|queen))
```

the captured substrings are “red king”, “red”, and “king”, and are numbered 1, 2, and 3.

The fact that plain parentheses fulfil two functions is not always helpful. There are often times when a grouping subpattern is required without a capturing requirement. If an opening parenthesis is followed by "?:", the subpattern does not do any capturing, and is not counted when computing the number of any subsequent capturing subpatterns. For example, if the string "the white queen" is matched against the pattern

```
the ((?:red|white) (king|queen))
```

the captured substrings are "white queen" and "queen", and are numbered 1 and 2. The maximum number of captured substrings is 99, and the maximum number of all subpatterns, both capturing and non-capturing, is 200.

As a convenient shorthand, if any option settings are required at the start of a non-capturing subpattern, the option letters may appear between the "?" and the ":". Thus the two patterns

```
(?i:saturday|sunday)
```

```
(?:(?i)saturday|sunday)
```

match exactly the same set of strings. Because alternative branches are tried from left to right, and options are not reset until the end of the subpattern is reached, an option setting in one branch does affect subsequent branches, so the above patterns match "SUNDAY" as well as "Saturday".

REPETITION

Repetition is specified by quantifiers, which can follow any of the following items:

a single character, possibly escaped

the . metacharacter

a character class

a back reference (see next section)

a parenthesized subpattern (unless it is an assertion - see below)

The general repetition quantifier specifies a minimum and maximum number of permitted matches, by giving the two numbers in curly brackets (braces), separated by a comma. The numbers must be less than 65536, and the first must be less than or equal to the second. For example:

```
z{2,4}
```

matches "zz", "zzz", or "zzzz". A closing brace on its own is not a special character. If the second number is omitted, but the comma is present, there is no upper limit; if the second number and the comma are both omitted, the quantifier specifies an exact number of required matches. Thus

```
[aeiou]{3,}
```

matches at least 3 successive vowels, but may match many more, while

```
\d{8}
```

matches exactly 8 digits. An opening curly bracket that appears in a position where a quantifier is not allowed, or one that does not match the syntax of a quantifier, is taken as a literal character. For example, {,6} is not a quantifier, but a literal string of four characters.

The quantifier {0} is permitted, causing the expression to behave as if the previous item and the quantifier were not present.

For convenience (and historical compatibility) the three most common quantifiers have single-character abbreviations:

```
*   is equivalent to {0,}

+   is equivalent to {1,}

?   is equivalent to {0,1}
```

It is possible to construct infinite loops by following a subpattern that can match no characters with a quantifier that has no upper limit, for example:

```
(a?)*
```

Earlier versions of Perl and PCRE used to give an error at compile time for such patterns. However, because there are cases where this can be useful, such patterns are now accepted, but if any repetition of the subpattern does in fact match no characters, the loop is forcibly broken.

By default, the quantifiers are “greedy”, that is, they match as much as possible (up to the maximum number of permitted times), without causing the rest of the pattern to fail. The classic example of where this gives problems is in trying to match comments in C programs. These appear between the sequences /* and */ and within the sequence, individual * and / characters may appear. An attempt to match C comments by applying the pattern

```
/\*.*\*/
```

to the string

```
/* first command */  not comment  /* second comment */
```

fails, because it matches the entire string due to the greediness of the .* item.

However, if a quantifier is followed by a question mark, then it ceases to be greedy, and instead matches the minimum number of times possible, so the pattern

```
/\*.*?\*/
```

does the right thing with the C comments. The meaning of the various quantifiers is not otherwise changed, just the preferred number of matches. Do not confuse this use of question mark with its use as a quantifier in its own right. Because it has two uses, it can sometimes appear doubled, as in

```
\d??\d
```

which matches one digit by preference, but can match two if that is the only way the rest of the pattern matches.

If the PCRE_UNGREEDY option is set (an option which is not available in Perl) then the quantifiers are not greedy by default, but individual ones can be made greedy by following them with a question mark. In other words, it inverts the default behaviour.

When a parenthesized subpattern is quantified with a minimum repeat count that is greater than 1 or with a limited maximum, more store is required for the compiled pattern, in proportion to the size of the minimum or maximum.

If a pattern starts with .* or .{0,} and the PCRE_DOTALL option (equivalent to Perl's /s) is set, thus allowing the . to match newlines, then the pattern is implicitly anchored, because whatever follows will be tried against every character position in the subject string, so there is no point in retrying the overall match at any position after the first. PCRE treats such a pattern as though it were preceded by \A. In cases where it is known that the subject string contains no newlines, it is worth setting PCRE_DOTALL when the pattern begins with .* in order to obtain this optimization, or alternatively using ^ to indicate anchoring explicitly.

When a capturing subpattern is repeated, the value captured is the substring that matched the final iteration. For example, after

```
  (tweedle[dume]{3}\s*)+
```

has matched "tweedledum tweedledee" the value of the captured substring is "tweedledee". However, if there are nested capturing subpatterns, the corresponding captured values may have been set in previous iterations. For example, after

```
  /(a|(b))+/
```

matches "aba" the value of the second captured substring is "b".

BACK REFERENCES

Outside a character class, a backslash followed by a digit greater than 0 (and possibly further digits) is a back reference to a capturing subpattern earlier (i.e. to its left) in the pattern, provided there have been that many previous capturing left parentheses.

However, if the decimal number following the backslash is less than 10, it is always taken as a back reference, and causes an error only if there are not that many capturing left parentheses in the entire pattern. In other words, the parentheses that are referenced need not be to the left of the reference for numbers less than 10. See the section entitled "Backslash" above for further details of the handling of digits following a backslash.

A back reference matches whatever actually matched the capturing subpattern in the current subject string, rather than anything matching the subpattern itself. So the pattern

```
  (sens|respons)e and \1ibility
```

matches "sense and sensibility" and "response and responsibility", but not "sense and responsibility". If caseful matching is in force at the time of the back reference, then the case of letters is relevant. For example,

```
  ((?i)rah)\s+\1
```

matches “rah rah” and “RAH RAH”, but not “RAH rah”, even though the original capturing subpattern is matched caselessly.

There may be more than one back reference to the same subpattern. If a subpattern has not actually been used in a particular match, then any back references to it always fail. For example, the pattern

```
(a|(bc))\2
```

always fails if it starts to match “a” rather than “bc”. Because there may be up to 99 back references, all digits following the backslash are taken as part of a potential back reference number. If the pattern continues with a digit character, then some delimiter must be used to terminate the back reference. If the PCRE_EXTENDED option is set, this can be whitespace. Otherwise an empty comment can be used.

A back reference that occurs inside the parentheses to which it refers fails when the subpattern is first used, so, for example, (a\1) never matches. However, such references can be useful inside repeated subpatterns. For example, the pattern

```
(a|b\1)+
```

matches any number of “a”s and also “aba”, “ababaa” etc. At each iteration of the subpattern, the back reference matches the character string corresponding to the previous iteration. In order for this to work, the pattern must be such that the first iteration does not need to match the back reference. This can be done using alternation, as in the example above, or by a quantifier with a minimum of zero.

ASSERTIONS

An assertion is a test on the characters following or preceding the current matching point that does not actually consume any characters. The simple assertions coded as \b, \B, \A, \Z, \z, ^ and $ are described above. More complicated assertions are coded as subpatterns. There are two kinds: those that look ahead of the current position in the subject string, and those that look behind it.

An assertion subpattern is matched in the normal way, except that it does not cause the current matching position to be changed. Lookahead assertions start with (?= for positive assertions and (?! for negative assertions. For example,

```
\w+(?=;)
```

matches a word followed by a semicolon, but does not include the semicolon in the match, and

```
foo(?!bar)
```

matches any occurrence of “foo” that is not followed by “bar”. Note that the apparently similar pattern

```
(?!foo)bar
```

does not find an occurrence of “bar” that is preceded by something other than “foo”; it finds any occurrence of “bar” whatsoever, because the assertion (?!foo) is always true when the next three characters are “bar”. A lookbehind assertion is needed to achieve this effect.

Look-behind assertions start with (?<= for positive assertions and (?<! for negative assertions. For example,

```
  (?<!foo)bar
```

does find an occurrence of “bar” that is not preceded by “foo”. The contents of a lookbehind assertion are restricted such that all the strings it matches must have a fixed length. However, if there are several alternatives, they do not all have to have the same fixed length. Thus

```
  (?<=bullock|donkey)
```

is permitted, but

```
  (?<!dogs?|cats?)
```

causes an error at compile time. Branches that match different length strings are permitted only at the top level of a lookbehind assertion. This is an extension compared with Perl 5.005, which requires all branches to match the same length of string. An assertion such as

```
  (?<=ab(c|de))
```

is not permitted, because its single top-level branch can match two different lengths, but it is acceptable if rewritten to use two top-level branches:

```
  (?<=abc|abde)
```

The implementation of lookbehind assertions is, for each alternative, to temporarily move the current position back by the fixed width and then try to match. If there are insufficient characters before the current position, the match is deemed to fail. Lookbehinds in conjunction with once-only subpatterns can be particularly useful for matching at the ends of strings; an example is given at the end of the section on once-only subpatterns.

Several assertions (of any sort) may occur in succession. For example,

```
  (?<=\d{3})(?<!999)foo
```

matches “foo” preceded by three digits that are not “999”. Notice that each of the assertions is applied independently at the same point in the subject string. First there is a check that the previous three characters are all digits, then there is a check that the same three characters are not “999”. This pattern does not match “foo” preceded by six characters, the first of which are digits and the last three of which are not “999”. For example, it doesn’t match “123abcfoo”. A pattern to do that is

```
  (?<=\d{3}...)(?<!999)foo
```

This time the first assertion looks at the preceding six characters, checking that the first three are digits, and then the second assertion checks that the preceding three characters are not “999”.

Assertions can be nested in any combination. For example,

```
(?<=(?<!foo)bar)baz
```

matches an occurrence of “baz” that is preceded by “bar” which in turn is not preceded by “foo”, while

```
(?<=\d{3}(?!999)...)foo
```

is another pattern which matches “foo” preceded by three digits and any three characters that are not “999”.

Assertion subpatterns are not capturing subpatterns, and may not be repeated, because it makes no sense to assert the same thing several times. If any kind of assertion contains capturing subpatterns within it, these are counted for the purposes of numbering the capturing subpatterns in the whole pattern. However, substring capturing is carried out only for positive assertions, because it does not make sense for negative assertions.

Assertions count towards the maximum of 200 parenthesized subpatterns.

ONCE-ONLY SUBPATTERNS

With both maximizing and minimizing repetition, failure of what follows normally causes the repeated item to be re-evaluated to see if a different number of repeats allows the rest of the pattern to match. Sometimes it is useful to prevent this, either to change the nature of the match, or to cause it fail earlier than it otherwise might, when the author of the pattern knows there is no point in carrying on.

Consider, for example, the pattern \d+foo when applied to the subject line

```
123456bar
```

After matching all 6 digits and then failing to match “foo”, the normal action of the matcher is to try again with only 5 digits matching the \d+ item, and then with 4, and so on, before ultimately failing. Once-only subpatterns provide the means for specifying that once a portion of the pattern has matched, it is not to be re-evaluated in this way, so the matcher would give up immediately on failing to match “foo” the first time. The notation is another kind of special parenthesis, starting with (?> as in this example:

```
(?>\d+)bar
```

This kind of parenthesis “locks up” the part of the pattern it contains once it has matched, and a failure further into the pattern is prevented from backtracking into it. Backtracking past it to previous items, however, works as normal.

An alternative description is that a subpattern of this type matches the string of characters that an identical standalone pattern would match, if anchored at the current point in the subject string.

Once-only subpatterns are not capturing subpatterns. Simple cases such as the above example can be thought of as a maximizing repeat that must swallow everything it can. So, while both \d+ and \d+? are prepared to adjust the number of digits they match in order to make the rest of the pattern match, (?>\d+) can only match an entire sequence of digits.

This construction can of course contain arbitrarily complicated subpatterns, and it can be nested.

Once-only subpatterns can be used in conjunction with look-behind assertions to specify efficient matching at the end of the subject string. Consider a simple pattern such as

```
  abcd$
```

when applied to a long string which does not match it. Because matching proceeds from left to right, PCRE will look for each “a” in the subject and then see if what follows matches the rest of the pattern. If the pattern is specified as

```
  ^.*abcd$
```

then the initial .* matches the entire string at first, but when this fails, it backtracks to match all but the last character, then all but the last two characters, and so on. Once again the search for “a” covers the entire string, from right to left, so we are no better off. However, if the pattern is written as

```
  ^(?>.*)(?<=abcd)
```

then there can be no backtracking for the .* item; it can match only the entire string. The subsequent lookbehind assertion does a single test on the last four characters. If it fails, the match fails immediately. For long strings, this approach makes a significant difference to the processing time.

CONDITIONAL SUBPATTERNS

It is possible to cause the matching process to obey a subpattern conditionally or to choose between two alternative subpatterns, depending on the result of an assertion, or whether a previous capturing subpattern matched or not. The two possible forms of conditional subpattern are

```
  (?(condition)yes-pattern)

  (?(condition)yes-pattern|no-pattern)
```

If the condition is satisfied, the yes-pattern is used; otherwise the no-pattern (if present) is used. If there are more than two alternatives in the subpattern, a compile-time error occurs.

There are two kinds of condition. If the text between the parentheses consists of a sequence of digits, then the condition is satisfied if the capturing subpattern of that number has previously matched. Consider the following pattern, which contains non-significant white space to make it more readable (assume the PCRE_EXTENDED option) and to divide it into three parts for ease of discussion:

```
  ( \( )?    [^()]+    (?(1) \) )
```

The first part matches an optional opening parenthesis, and if that character is present, sets it as the first captured substring. The second part matches one or more characters that are not parentheses. The third part is a conditional subpattern that tests whether the first set of parentheses matched or not. If they did, that is, if subject started with an opening parenthesis, the condition is true, and so the yes-pattern is executed and a closing parenthesis is required. Otherwise, since no-pattern is not present, the subpattern matches nothing.

In other words, this pattern matches a sequence of non-parentheses, optionally enclosed in parentheses.

If the condition is not a sequence of digits, it must be an assertion. This may be a positive or negative lookahead or lookbehind assertion. Consider this pattern, again containing non-significant white space, and with the two alternatives on the second line:

```
  (?(?=[^a-z]*[a-z])

  \d{2}[a-z]{3}-\d{2}  |  \d{2}-\d{2}-\d{2} )
```

The condition is a positive lookahead assertion that matches an optional sequence of non-letters followed by a letter. In other words, it tests for the presence of at least one letter in the subject. If a letter is found, the subject is matched against the first alternative; otherwise it is matched against the second. This pattern matches strings in one of the two forms dd-aaa-dd or dd-dd-dd, where aaa are letters and dd are digits.

COMMENTS

The sequence (?# marks the start of a comment which continues up to the next closing parenthesis. Nested parentheses are not permitted. The characters that make up a comment play no part in the pattern matching at all.

If the PCRE_EXTENDED option is set, an unescaped # character outside a character class introduces a comment that continues up to the next newline character in the pattern.

PERFORMANCE

Certain items that may appear in patterns are more efficient than others. It is more efficient to use a character class like [aeiou] than a set of alternatives such as (a|e|i|o|u). In general, the simplest construction that provides the required behaviour is usually the most efficient. Jeffrey Friedl's book contains a lot of discussion about optimizing regular expressions for efficient performance.

When a pattern begins with .* and the PCRE_DOTALL option is set, the pattern is implicitly anchored by PCRE, since it can match only at the start of a subject string. However, if PCRE_DOTALL is not set, PCRE cannot make this optimization, because the . metacharacter does not then match a newline, and if the subject string contains newlines, the pattern may match from the character immediately following one of them instead of from the very start. For example, the pattern

```
  (.*) second
```

matches the subject "first\nand second" (where \n stands for a newline character) with the first captured substring being "and". In order to do this, PCRE has to retry the match starting after every newline in the subject.

If you are using such a pattern with subject strings that do not contain newlines, the best performance is obtained by setting PCRE_DOTALL, or starting the pattern with ^.* to indicate explicit anchoring. That saves PCRE from having to scan along the subject looking for a newline to restart at.

Beware of patterns that contain nested indefinite repeats. These can take a long time to run when applied to a string that does not match. Consider the pattern fragment

```
(a+)*
```

This can match "aaaa" in 33 different ways, and this number increases very rapidly as the string gets longer. (The * repeat can match 0, 1, 2, 3, or 4 times, and for each of those cases other than 0, the + repeats can match different numbers of times.) When the remainder of the pattern is such that the entire match is going to fail, PCRE has in principle to try every possible variation, and this can take an extremely long time.

An optimization catches some of the more simple cases such as

```
(a+)*b
```

where a literal character follows. Before embarking on the standard matching procedure, PCRE checks that there is a "b" later in the subject string, and if there is not, it fails the match immediately. However, when there is no following literal this optimization cannot be used. You can see the difference by comparing the behaviour of

```
(a+)*\d
```

with the pattern above. The former gives a failure almost instantly when applied to a whole line of "a" characters, whereas the latter takes an appreciable time with strings longer than about 20 characters.

AUTHOR

Philip Hazel <ph10@cam.ac.uk>
University Computing Service,
New Museums Site,
Cambridge CB2 3QG, England.
Phone: +44 1223 334714
Last updated: 29 July 1999

Undefined variables and dynamic loading

During execution, should the ICI execution engine fail to find a variable within the current scope, it will attempt to load a library based on the name of that variable. Such a library may be a host specific dynamically loaded native machine code library, an ICI module, or both.

In attempting to load an ICI module, a file name of the form:

ici*var*.ici

is considered, where *var* is the as yet undefined variable name. This file is searched for on the current host specific search path. If found, a new extern, static and auto scope is established and the new extern scope struct is assigned to *var* in the outermost writable scope available. That outermost writable scope also forms the super of the new extern scope. The module is then parsed with the given scope, after which the variable lookup is repeated. In normal practice this will mean that the loaded module has an outer scope holding all the normal ICI primitives and a new empty extern scope. The intent of this mechanism is that the loaded module should define all its published functions in its extern scope. References by an invoking program to functions and other objects of the loaded module would always be made explicitly through the *var* which references the module. For example, a program might contain the fragment:

```
query = cgi.decode_query();
cgi.start_page(“Query results”);
```

where “cgi” is undefined, but the file *icicgi.ici* exists on the search path and includes function definitions such as:

```
extern
decode_query()
{
    ...
}

extern
start_page(title)
{
    ...
}
```

Upon first encountering the variable *cgi* in the code fragment the module *icicgi.ici* will be parsed and its extern scope assigned to the new variable *cgi* in the outermost scope of the program (that is, the most global scope). The lookup of the variable *cgi* is then repeated, this time finding the structure which contains the function *decode_query*. The second, and all subsequent, use of the variable *cgi* will be satisfied immediately from the already loaded module.

In attempting to load a host specific dynamically loaded native machine code library, a file name of the form:

`ici`*`var`*`.`*`ext`*

is considered, where *var* is the as yet undefined variable name and *ext* is the normal host extension for such libraries. This file is searched for on the current host specific search path. If found the file is loaded into the ICI interpreter’s address space using the local host’s dynamic library loading mechanism. An initialisation function in the loaded library may return an ICI object (see below). Should an object be returned, it is assigned to *var* in the outermost writable scope available. Further, should the returned variable be a structure, additional loading of an ICI module of the same name is allowed (as described above) and the returned struct forms the structure for externs in that load.

The basics of writing dynamic loading native machine code modules

This description is bare-bones and assumes a knowledge of ICI’s internals.

The loaded library must contain a function of the following name and declaration:

```
object_t *
ici_var_library_init()
```

```
{
    ...
}
```

where *var* is the as yet undefined variable name. This is the initialisation function which is called when the library is loaded. This function should return an ICI object, or NULL on error, in which case the ICI error variable must be set. The returned object will be assigned to *var* as described above.

Modules of the dynamically loaded library which include ICI header files must have the directory holding the ICI header files on their include search path and have two preprocessor definitions established before any of the ICI headers are included (they are typically defined in the makefile or project settings). These are:

CONFIG_FILE	Which must be defined to be the name of the ICI configuration file which is specific to this installation. The defined value should include double quotes around the name. For example: `“conf-w32.h”` is the file used by Windows, and this would be defined on the command line or in the project settings with: `/DCONFIG_FILE=\”conf-w32.h\”`
ICI_DLL	Which must simply be defined. This causes certain changes in the nature of data declarations in the ICI header files which are required on some systems (such as Windows) to allow imported data references.

The following sample module, *mbox.c*, illistrates a typical form for a simple dynamically loaded ICI module.

```
#include <windows.h>
#include "func.h"
#include "struct.h"

/*
 * mbox_msg => mbox.msg(string) from ICI
 *
 * Pops up a modal message box with the given string in it and waits for the
 * use to hit OK. Returns NULL.
 */
int
mbox_msg()
{
    char    *msg;

    if (typecheck("s", &msg))
      return 1;
    MessageBox(NULL, msg, (LPCTSTR)"ICI", MB_OK | MB_SETFOREGROUND);
    return null_ret();
}

/*
 * Object stubs for our intrinsic functions.
 */
cfunc_tmbox_cfuncs[] =
{
    {CF_OBJ,"msg",mbox_msg},
    {CF_OBJ}
};

/*
 * ici_mbox_library_init
 *
 * Initialisation routine called on load of this module into the ICI
 * interpreters address space. Creates and returns a struct which will
 * be assigned to "mbox". This struct contains references to our
 * intrinsic functions.
 */
object_t *
ici_mbox_library_init()
{
    struct_t*s;

    if ((s = new_struct()) == NULL)
      return NULL;
    if (ici_assign_cfuncs(s, mbox_cfuncs))
      return NULL;
    return objof(s);
}
```

The following simple Makefile illustrates forms suitable for compiling this module into a DLL under Windows. Note in particular the defines in the CFLAGS and the use of /export in the link line to make the function *ici_mbox_library_init* externally visible.

```
CFLAGS= -I.. /DCONFIG_FILE=\"conf-w32.h\" /DICI_DLL

OBJS  = mbox.obj
LIBS  = ../ici.lib user32.lib

icimbox.dll: $(OBJS)
            link /dll /out:$@ $(OBJS) /export:ici_mbox_library_init $(LIBS)
```

Note that there is no direct supprt for the /export option in the MS Developer Studio link settings panel, but it can be entered directly in the *Project Options* text box.

The following Makefile achieves an equivalent result under Solaris:

```
CC    = gcc -pipe -g
CFLAGS= -fpic -I.. -DCONFIG_FILE='"conf-sun.h"' -DICI_DLL

OBJS  = mbox.o

icimbox.so  : $(OBJS)
            ld -o $@ -dc -dp $(OBJS)
```