Inform
in four minutes

A quick reference to the Inform programming language

Version 1.2 (January 2001)

The road to brevity is via imprecision and through solecism – refer to the *Inform Designer's Manual* for the definitive story.

●● Literals

A Z-code **word** literal uses sixteen bits (whereas a Glulx word has thirty-two bits). A **byte** literal is always eight bits.

- Decimal: `-32768` to `32767`
  Hexadecimal: `$0` to `$FFFF`
  Binary: `$$0` to `$$1111111111111111`
- Action: `##Look`
- Character: `'a'`
- Dictionary word: `'aardvark'` (up to nine characters significant); use circumflex "^" to denote apostrophe.
  Plural word: `'aardvarks//p'`
  Single-character word: `"a"` (`name` property only) or `'a//'`
- String: `"aardvark's adventure"` (maximum around 4000 characters); can include special values including:

`^`	newline
`~`	quotes "`"`"
`@@64`	at sign "@"
`@@92`	backslash "\"
`@@94`	circumflex "^"
`@@126`	tilde "~"
``@`a``	a with a grave accent "à", et al
`@LL`	pound sign "£", et al
`@00` ... `@32`	low string 0..32

●● Names

The identifier of an Inform *`constant`*, *`variable`*, *`array`*, *`class`*, *`object`*, *`property`*, *`attribute`*, *`routine`* or *`label`*. Up to 32 characters: alphabetic (case not significant), numeric and underscore, with the first character not a digit.

●● Expressions and Operators

Use parentheses `(...)` to control the order of evaluation.

Arithmetic/logical expressions support these operators:

`p + q`	addition
`p - q`	subtraction
`p * q`	multiplication
`p / q`	integer division
`p % q`	remainder
`p++`	increments *p*, evaluates to original value
`++p`	increments *p*, evaluates to new value
`p--`	decrements *p*, evaluates to original value
`--p`	decrements *p*, evaluates to new value
`p & q`	bitwise AND
`p | q`	bitwise OR
`~p`	bitwise NOT (inversion)

Conditional expressions return `true` (1) or `false` (0); *q* may be a list of alternatives *`q1`* `or` *`q2`* `or ...` *`qN`*:

`p == q`	*p* is equal to *q*
`p ~= q`	*p* isn't equal to *q*
`p > q`	*p* is greater than *q*
`p < q`	*p* is less than *q*
`p >= q`	*p* is greater than or equal to *q*
`p <= q`	*p* is less than or equal to *q*
`p ofclass q`	object *p* is of class *q*
`p in q`	object *p* is a child of object *q*
`p notin q`	object *p* isn't a child of object *q*
`p provides q`	object *p* provides property *q*
`p has q`	object *p* has attribute *q*
`p hasnt q`	object *p* hasn't attribute *q*

Boolean expressions return `true` (1) or `false` (0):

`p && q`	both *p* and *q* are true (non-zero)
`p || q`	either *p* or *q* is true (non-zero)
`~~p`	*p* is false (zero)

To return –1, 0 or 1 based on unsigned comparison:

```
UnsignedCompare(p,q)
```

To return `true` if object *q* is a child or grand-child or... of *p*:

```
IndirectlyContains(p,q)
```

To return a random number 1..*N*, or a value from a list:

```
random(N)
random(value,value, ... value)
```

●● Constants

Named word values, unchanging at run-time, which are by default initialised to zero:

```
Constant constant;
Constant constant = expr;
```

Standard constants are `true` (1), `false` (0) and `nothing` (0), also `null` (–1).

To define a constant (unless it already exists):

```
Default constant expr;
```

●● Variables and Arrays

Named word/byte values which can change at run-time and are by default initialised to zero.

A **global** variable is a single word:

```
Global variable;
Global variable = expr;
```

A **word array** is a set of global words accessed using *`array`*`-->0`, *`array`*`-->1`, ... *`array`*`-->(`*`N`*`-1)`:

```
Array array --> N;
Array array --> expr1 expr2 ... exprN;
Array array --> "string";
```

A **table array** is a set of global words accessed using *`array`*`-->1`, *`array`*`-->2`, ... *`array`*`-->`*`N`*, with *`array`*`-->0` initialised to *N*:

```
Array array table N;
Array array table expr1 expr2 ... exprN;
Array array table "string";
```

A **byte array** is a set of global bytes accessed using *`array`*`->0`, *`array`*`->1`, ... *`array`*`->(`*`N`*`-1)`:

```
Array array -> N;
Array array -> expr1 expr2 ... exprN;
Array array -> "string";
```

A **string array** is a set of global bytes accessed using *`array`*`->1`, *`array`*`->2`, ... *`array`*`->`*`N`*, with *`array`*`->0` initialised to *N*:

```
Array array string N;
Array array string expr1 expr2 ... exprN;
Array array string "string";
```

In all these cases, the characters of the initialising *`string`* are unpacked to the individual word/byte elements of the array.

See also Objects (for **property** variables) and Routines (for **local** variables).

Classes and Objects

To declare a *class* – a template for a family of objects – where the optional (*N*) limits instances created at run-time:

```
Class   class(N)
  class class class ... class
  has   attr_def attr_def ... attr_def
  with  prop_def,
        ...
        prop_def;
```

To declare an *object*; "`Object`" can instead be a *class*, the remaining four header items are all optional, and *arrows* (`->`, `-> ->`, ...) and *parent_object* are incompatible:

```
Object   arrows object "ext_name" parent_object
  class class class ... class
  has   attr_def attr_def ... attr_def
  with  prop_def,
        ...
        prop_def;
```

The `class`, `has` and `with` (and also the rarely-used `private`) segments are all optional, and can appear in any order.

To determine an object's class as one of `Class`, `Object`, `Routine`, `String` (or `nothing`):

```
metaclass(object)
```

has segment: Each *attr_def* is either of:

```
attribute
~attribute
```

To change attributes at run-time:

```
give object attr_def attr_def ... attr_def;
```

with/private segments: Each *prop_def* declares a variable (or word array) and can take any of these forms (where a *value* is an expression, a string or an embedded routine):

```
property
property value
property value value ... value
```

A property variable is addressed by *object.property* (or within the object's declaration as `self`.*property*) .

Multiple *values* create a property array; in this case *object*.#*property* is the number of **bytes** occupied by the array, the entries can be accessed using *object*.&*property*-->0, *object*.&*property*-->1, ... , and *object.property* refers to the value of the first entry.

A property variable inherited from an object's class is addressed by *object.class*::*property*; this gives the original value prior to any changes within the object.

Manipulating the object tree

To change object relationships at run-time:

```
move object to parent_object;
remove object;
```

To return the parent of an object (or `nothing`):

```
parent(object)
```

To return the first child of an object (or `nothing`):

```
child(object)
```

To return the adjacent child of an object's parent (or `nothing`):

```
sibling(object)
```

To return the number of child objects directly below an object:

```
children(object)
```

Message passing

To a class:

```
class.remaining()
class.create()
class.destroy(object)
class.recreate(object)
class.copy(from_object,to_object)
```

To an object:

```
object.property(a1,a2, ... a7)
```

To a routine:

```
routine.call(a1,a2, ... a7)
```

To a string:

```
string.print()
string.print_to_array(array)
```

Statements

Each *statement* is terminated by a semi-colon "`;`".
A *statement_block* is a single *statement* or a series of *statements* enclosed in braces `{...}`.

An exclamation "`!`" starts a comment – rest of line ignored.

A common statement is the assignment:

```
variable = expr;
```

There are two forms of multiple assignment:

```
variable = variable = ... = expr;
variable = expr, variable = expr, ... ;
```

Routines

A routine can have any number of **local variables**: word values which are private to the routine and which by default are set to zero on each call. Recursion is permitted.

A **standalone** routine:

- has a name, by which it is called using *routine*`()`; can also be called indirectly using `indirect(`*routine*,*a1*,*a2*, ... *a7*`)`
- can take arguments, using *routine*(*a1*,*a2*, ... *a7*), whose values initialise the equivalent local variables
- returns `true` at the final "`]`"

```
[ routine
    local_var local_var ... local_var;
    statement;
    statement;
    ...
    statement;
    ];
```

A routine **embedded** as the value of an object property:

- has no name, and is called when the property is invoked; can also be called explicitly using *object.property*`()`
- accepts arguments only when called explicitly
- returns `false` at the final "`]`"

```
property [
    local_var local_var ... local_var;
    statement;
    statement;
    ...
    statement;
    ]
```

Routines return a single value, when execution reaches the final "`]`" or an explicit return statement:

```
return expr;
return;
rtrue;
rfalse;
```

To define a dummy standalone routine with *N* local variables (unless it already exists):

```
Stub routine N;
```

Flow control

To execute statements if *expr* is `true`; optionally, to execute other statements if *expr* is `false`:

```
if (expr)
    statement_block
```

```
if (expr)
    statement_block
else
    statement_block
```

To execute statements depending on the value of *expr*:

```
switch (expr) {
    value: statement; ... statement;
    value: statement; ... statement;
    ...
    default: statement; ... statement;
    }
```

where each *value* can be given as:

```
constant
lo_constant to hi_constant
constant,constant, ... constant
```

Loop control

To execute statements while *expr* is `true`:

```
while (expr)
    statement_block
```

To execute statements until *expr* is `true`:

```
do
    statement_block
    until (expr)
```

To execute statements while a variable changes:

```
for (set_var : loop_while_expr : update_var)
    statement_block
```

To execute statements for all defined objects:

```
objectloop (variable)
    statement_block
```

To execute statements for all objects selected by *expr*:

```
objectloop (expr_starting_with_variable)
    statement_block
```

To jump out of the current innermost loop or switch:

```
break;
```

To immediately start the next iteration of the current loop:

```
continue;
```

Displaying information

To output a list of values:

```
print value,value, ... value;
```

To output a list of values followed by a newline, then return `true` from the current routine:

```
print_ret value,value, ... value;
```

If the first (or only) *value* is a string, “`print_ret`” can be omitted:

```
"string",value, ... value;
```

Each *value* can be an expression, a string or a rule.

An **expression** is output as a signed decimal value.

A **string** in quotes `"..."` is output as text.

A **rule** is one of:

`(number)` *expr*	the *expr* in words
`(char)` *expr*	the *expr* as a single character
`(string)` *addr*	the string at the *addr*
`(address)` *addr*	the dictionary word at the *addr*
`(name)` *object*	the external (short) name of the *object*
`(a)` *object*	the short name preceded by “`a/an`”
`(the)` *object*	the short name preceded by “`the`”
`(The)` *object*	the short name preceded by “`The`”
`(`*routine*`)` *value*	the output when calling *routine*`(`*value*`)`

To output a newline character:

```
new_line;
```

To output multiple spaces:

```
spaces expr;
```

To output text in a display box:

```
box "string" "string" ... "string";
```

To change from regular to fixed-pitch font:

```
font off;
...
font on;
```

To change the font attributes:

```
style bold;        ! use one or more of these
style underline;   !
style reverse;     !
...
style roman;
```

Uncommon and deprecated statements

To jump to a labelled statement:

```
jump label;
...
.label; statement;
```

To terminate the program:

```
quit;
```

To save and restore the program state:

```
save label;
...
restore label;
```

To output the Inform compiler version number:

```
inversion;
```

To accept data from the current input stream:

```
read text_array parse_array routine;
```

To assign to one of 32 ‘low string’ variables:

```
string N "string";
```

```
Lowstring string_var  "string";
string N string_var;
```

Verbs and Actions

To specify a new verb:

```
Verb 'verb' 'verb' ... 'verb'
     * token  token ... token -> action
     * token  token ... token -> action
     ...
     * token  token ... token -> action;
```

where instead "`Verb`" can be "`Verb meta`", "*`action`*" can be "*`action`* `reverse`"; *`tokens`* are optional and each is one of:

`'`*`word`*`'`	that literal word
`'`*`w1`*`'/'`*`w2`*`'/...`	any one of those literal words
`attribute`	an object with that attribute
`creature`	an object with `animate` attribute
`held`	an object held by the player
`noun`	an object in scope
`noun=`*`routine`*	an object for which *`routine`* returns `true`
`scope=`*`routine`*	an object in this re-definition of scope
`multiheld`	one or more objects held by the player
`multi`	one or more objects in scope
`multiexcept`	as multi, omitting the specified object
`multiinside`	as multi, omitting those in specified object
`topic`	any text
`number`	any number
`routine`	a general parsing routine

To add synonyms to an existing verb:

```
Verb 'verb' 'verb' ... = 'existing_verb';
```

To modify an existing verb:

```
Extend 'existing_verb' last
     * token  token ... token -> action
     * token  token ... token -> action
     ...
     * token  token ... token -> action;
```

where instead "`Extend`" can be "`Extend only`" and "`last`" can be omitted, or changed to "`first`" or "`replace`"

To explicitly trigger a defined action (both *`noun`* and *`second`* are optional, depending on the *`action`*):

```
<action noun second>;
```

To explicitly trigger a defined action, then return `true` from the current routine:

```
<<action noun second>>;
```

Other useful directives

To include a directive within a routine definition `[...]`, insert a hash "#" as its first character.

To conditionally compile:

```
Ifdef name;      ! use any one of these
Ifndef name;     !
Iftrue expr;     !
Iffalse expr;    !
   ...
Ifnot;
   ...
Endif;
```

To display a compile-time message:

```
Message "string";
```

To include the contents of a file, searching the Library path:

```
Include "source_file";
```

To include the contents of a file in the same location as the current file:

```
Include ">source_file";
```

To specify that a library routine is to be replaced:

```
Replace routine;
```

To set the game's release number (default is 1), serial number (default is today's *yymmdd*) and status line format (default is `score`):

```
Release expr;
Serial "yymmdd";
Statusline score;
Statusline time;
```

To declare a new attribute common to all objects:

```
Attribute attribute;
```

To declare a new property common to all objects:

```
Property property;
Property property expr;
```

Uncommon and deprecated directives

You're unlikely to need these; look them up if necessary.

```
Abbreviate "string" "string" ... "string";
End;
Import variable variable ... variable;
Link "compiled_file";
Switches list_of_compiler_switches;
System_file;
```

File structure

A minimal source file:

```
Constant Story "MYGAME";
Constant Headline "^My first Inform game.^";
Constant MANUAL_PRONOUNS;

Include "Parser";
Include "VerbLib";

[ Initialise; location = study; "^Hello!^"; ];

Class   Room
  with  description "A bare room."
  has   light;

Class   Furniture
  with  before [; Take,Pull,Push,Pushdir:
            print_ret (The) self,
                " is too heavy for that."; ]
  has   static supporter;

Room    study "Your study";

Furniture "writing desk" study
  with  name 'writing' 'desk' 'table';

Object  -> -> axe "rusty axe"
  with  name 'rusty' 'blunt' 'axe' 'hatchet',
        description "It seems old and blunt.";

Include "Grammar";
```

Compiler

To compile (on a PC, use "`infrmw32`" at the DOS prompt):

```
inform commands source_file
```

Useful *`commands`* include:

`-~S`	disable both Strict checks and Debug tools
`-~SD`	disable Strict checks, enable Debug tools
`-X`	enable Infix debugger
`-r`	output all game text to file (for spell-check)
`-s`	display game's size and other statistics
`-z`	display game's memory map
`-v8`	compile in Version 8 format (default is v5)
`+`*`dir`*`,`*`dir`*`,...`	search for Included files in these directories

To display full compiler help, type:

```
inform -h -h1 -h2
```