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

## Expressions and Operators

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

Arithmetic/logical expressions support these operators:

- **Addition**: `p + q`
- **Subtraction**: `p - q`
- **Multiplication**: `p * q`
- **Division**: `p / q`
- **Remainder**: `p % q`
- **Increment**: `p++`
- **Decrement**: `p--`
- **Bitwise AND**: `p & q`
- **Bitwise OR**: `p | q`
- **Bitwise NOT**: `~p`
- **Conditional expressions** return `true` (1) or `false` (0):
  - `p == q`
  - `p != q`
  - `p > q`
  - `p >= q`
  - `p < q`
  - `p <= q`
  - `p && q`
  - `p || q`
  - `~p`

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

- `p && q`
- `p || q`
- `~p`

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

- `p == q`
- `p != q`
- `p > q`
- `p >= q`
- `p < q`
- `p <= q`

String: “aardvark”’s adventure” (maximum around 4000 characters), can include special values including:

- `\n` newline
- `\'` single quote
- `\"` double quote
- `\` backslash
- `\0` null
- `\$` dollar sign
- `\£` pound sign

---

## 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/value values which can change at run-time and are by default initialised to zero.

A `global` variable is a single word:

- `Global variable = expr;`

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

- `Array array -> N;`
- `Array array -> expr 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 expr 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 -> expr 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 expr 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).

---

## Expressions and Operators

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

Arithmetic/logical expressions support these operators:

- **Addition**: `p + q`
- **Subtraction**: `p - q`
- **Multiplication**: `p * q`
- **Division**: `p / q`
- **Remainder**: `p % q`
- **Increment**: `p++`
- **Decrement**: `p--`
- **Bitwise AND**: `p & q`
- **Bitwise OR**: `p | q`
- **Bitwise NOT**: `~p`

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

- `p == q`
- `p != q`
- `p > q`
- `p >= q`
- `p < q`
- `p <= q`
- `p && q`
- `p || q`
- `~p`

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

- `p && q`
- `p || q`
- `~p`

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

- `p == q`
- `p != q`
- `p > q`
- `p >= q`
- `p < q`
- `p <= q`
- `p && q`
- `p || q`
- `~p`

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 one from a list of constant values:

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

---

## Names

The identifier of an Inform `constant, variable, array, class, object, property, attribute, routine` or `label`.

Up to 32 characters: alphanumeric (case not significant), numeric and underscore, with the first character not a digit.

---

## 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**: $550$ to $1111111111111111$
- **Action**: #bLook
- **Character**: ‘\’
- **Dictionary word**: ‘aardvark’ (up to nine characters significant); use circumflex “~” to denote apostrophe.
- **Plural word**: “aardvarks”/p

A `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
  - `@31` low string 0..31
To declare a class – a template for a family of objects – where the optional (N) limits instances created at run-time:

```c
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:

```c
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):

```c
    metaclass(object)
```

**has segment:** Each attr_def is either of:

- attribute
- attribute =

To change attributes at run-time:

```c
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):

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

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

Multiple `value` creates 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--6`, `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.

A routine can have up to 15 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 “}”

```c
[ 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 “}”

```c
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:

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

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

```c
    Stub routine N;
```
● Flow control
To execute statements if \( \text{expr} \) is true; optionally, to execute other statements if \( \text{expr} \) is false:
\[
\begin{align*}
\text{if (} & \text{expr}\text{)} \\
& \text{statement}_\text{block} \\
\text{if (} & \text{expr}\text{)} \\
& \text{statement}_\text{block} \\
\text{else} \\
& \text{statement}_\text{block}
\end{align*}
\]
To execute statements depending on the value of \( \text{expr} \):
\[
\text{switch (} \text{expr} \text{)} 
\begin{cases}
\text{value: } & \text{statement} \\
\text{... default: } & \text{statement}
\end{cases}
\]
where each \text{value} can be given as:
- \text{constant}
- \text{lo_constant to hi_constant}
- \text{constant.constant, ... constant}

● Loop control
To execute statements while \( \text{expr} \) is true:
\[
\text{while (} \text{expr} \text{)} \\
\text{statement}_\text{block}
\]
To execute statements until \( \text{expr} \) is true:
\[
\text{do} \\
\text{statement}_\text{block} \\
\text{until (} \text{expr} \text{)}
\]
To execute statements while a variable changes:
\[
\text{for (set_var : loop_while_expr : update_var)} \\
\text{statement}_\text{block}
\]
To execute statements for all defined objects:
\[
\text{objectloop (variable)} \\
\text{statement}_\text{block}
\]
To execute statements for all objects selected by \( \text{expr} \):
\[
\text{objectloop (expr_starting_with_variable)} \\
\text{statement}_\text{block}
\]
To jump out of the current innermost loop or switch:
\[
\text{break;}
\]
To immediately start the next iteration of the current loop:
\[
\text{continue;}
\]

● Displaying information
To output a list of values:
\[
\text{print value, value, ... value;}
\]
To output a list of values followed by a newline, then return true from the current routine:
\[
\text{print_ret value, value, ... value;}
\]
If the first (or only) \text{value} is a string, “print_ret” can be omitted:
\[
"\text{string}.value, ... value;"
\]
Each \text{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:
- \text{(number) expr} the expr in words
- \text{(char) expr} the expr as a single character
- \text{(string) addr} the string at the \text{addr}
- \text{(address) addr} the dictionary word at the \text{addr}
- \text{(name) object} the external (short) name of the \text{object}
- \text{(a) object} the short name preceded by “a/an”
- \text{(the) object} the short name preceded by “the”
- \text{(The) object} the short name preceded by “The”
- \text{(routine) value} the output when calling \text{routine(value)}

To output a newline character:
\[
\text{new_line;}
\]
To output multiple spaces:
\[
\text{spaces expr;}
\]
To output text in a display box:
\[
\text{box \"string\" \"string\" ... \"string\";}
\]
To change from regular to fixed-pitch font:
\[
\text{font off;}
\]
\[
\text{... font on;}
\]
To change the font attributes:
\[
\text{style bold;} \quad \text{! use one or more of these}
\]
\[
\text{style underline;} \quad \text{!}
\]
\[
\text{style reverse;} \quad \text{!}
\]
\[
\text{... style roman;}
\]

● Uncommon and deprecated statements
To jump to a labelled statement:
\[
\text{jump label:} \\
\text{.label; statement;}
\]
To terminate the program:
\[
\text{quit;}
\]
To save and restore the program state:
\[
\text{save label:} \\
\text{... restore label;}
\]
To output the Inform compiler version number:
\[
\text{inversion;}
\]
To accept data from the current input stream:
\[
\text{read text_array parse_array routine;}
\]
To assign to one of 32 ‘low string’ variables:
\[
\text{string N \"string\";}
\]
\[
\text{Lowstring string_var \"string\";}
\]
\[
\text{string N string_var;}
\]
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:

- 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
  - a general object
- noun=routine
  - an object for which routine returns true
- scope=routine
  - an object in this re-definition of scope
- multi
  - one or more objects held by the player
- multi=routine
  - one or more objects in scope
- multi=attribute
  - as multi, omitting the specified object
- multiinside
  - as multi, omitting those in specified object
- topic
  - any text
- number
  - any number
- routine
  - a general 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>>;
```