A BNF grammar of Red/System

Introduction

This grammar aims to describe, in a formalized way, the main syntactic and semantic properties of the Red/System language. The order of presentation of the language elements is systematic rather than tutorial. Thus it complements the document "Red/System Language Specifications" to be found at http://static.red-lang.org/red-system-specs-light.html, which is the authoritative source for Red/System. In case of discrepancies, it is that document which prevails, in combination with the actual text of the compiler, to be found at https://github.com/dockimbel/Red.

Explanations

For simplicity of presentation end-of-line comments are ignored. Also, the internal structure of lexical elements is not specified – all lexical elements are assumed to have been analysed and verified.

For explanatory purposes, some rules are repeated in several places. Moreover, the grammar is somewhat ambiguous; it is known, however, that the language can be analyzed in one pass by recursive descent. For completeness, the facilities for compile-time source manipulation (source file inclusion, conditional compilation and macro definition and expansion) are described in some detail, even though their treatment takes place before syntactic and semantic analysis, and in the rest of the description that treatment is assumed to have taken place already.

Each context-free rule (written in Courier New) is followed by semantic comments. Non-terminals of the grammar are referred to in these comments by being quoted in *italics*. When a semantic comment refers to the compilation order, this is to be understood as follows: all elements directly contained in the *program* are compiled first, in lexical order, including the function definitions (as regards the function name and the specification). Only then are the function bodies compiled, in the lexical order of occurrence of the function definitions.

Notational conventions:
- production rules are introduced by ::=
- non-terminals are enclosed in < >
- all other symbols in the rules stand for themselves, i.e. they are terminals (except the meta-symbols | { } ° * +, see next)
- alternatives are separated by |
- grouping is indicated by braces {} (this may contain alternatives*)
- optional elements are indicated by °
- repeating elements are indicated by * (zero or more times) and + (one or more times)
- some terminals are circumscribed by natural language in quotes ""

Author	Rudolf W. MEIJER
Document date	28-Nov-12
Red/System Language Specification	v. 37 of 24-Nov-12
Compiler source code version consulted	3ff5846 of 27-Nov-12

* These braces are not to be confused with the delimiters of a REBOL literal string!

Table of contents

1. program and directives

```
<program> ::= Red/System [ <meta-data> 0 ] <body-block>
<meta-data> ::= "not part of this specification"
<body-block> ::=
      [ {<directive> | <definition> | <statement> | <expression> | <comment>} * ]
```

A *program* is a top-level *body-block* preceded by a header which may contain *meta-data*. The other occurrences of *body-blocks* are in functions and contexts (see section 2). *Directives* (see below) and *definitions* (see section 2) may only occur at the level of a *body-block*. Within a *body-block*, *statements* and *expressions* may be mixed indiscriminately. The execution of a *statement* produces no value. The evaluation of an *expression* occurring directly within a *program* will produce a value but this will be discarded. In some other constructs where both *statements* and *expressions* are allowed, the value produced by the evaluation of an *expression* may or may not be discarded, as described below when discussing the relevant constructs. *Body-blocks* determine the scope of names defined in them, as described in section 3.

```
<comment> ::= comment {<expression> | [ <expression> + ]}
```

A *comment* is ignored by the compiler. Except when otherwise noted in what follows, it may occur wherever a *statement* or *expression* is allowed. Note that free form comments may be achieved by letting the expression be a string value.

```
<directive> ::= <source-manipulation-directive>
      | <enumeration-directive> | <interface-directive>
```

Source-manipulation-directives are treated by the compiler in a preliminary phase ("load time") before the syntactic and semantic analysis proper. Such *directives* may contain other directives, allowing a powerful control over the source text, which can be used, e.g. to have a single collection of source files addressing several target machine architectures and operating systems. The other *directives* are in fact specialized definition constructs, which are treated by the compiler along with other *definitions* proper.

```
<source-manipulation-directive> ::=
        <include-directive>
      | <conditional-compilation-directive>
      | <macro-directive>

<include-directive> ::= #include %<file>
<file> ::= "file name in REBOL format"
```

The *include-directive* will insert the text in the source file at the current position in the source text.

```
<conditional-compilation-directive> ::=
        #if <option-test> <body-block>
      | #either <option-test> <body-block> <body-block>
      | #switch <option-name> [ {<option-value> <body-block>} + ]
<option-test> ::= <option-name> <comparison-op> <option-value>
<option-name> ::= <word>
<option-value> ::= <integer> | <decimal> | <word> | '<word>
```

Conditional-compilation-directives insert the text contained within the selected *body-block* (not including the square brackets) at the current position in the source text, based on the comparison of the value assigned to the *option-name* with the specified *option-value*. The operation of the tests in the `#if`, `#either` and `#switch` variants should be self-explanatory. The assignment of a value to the *option-name* occurs when calling the Red/System compiler from the host operating system. The allowed *option-names* and *option-values* are documented in a configuration file

```
<macro-directive> ::= <simple-macro-directive> | <parametrized-macro-directive>
```

Macro-directives serve to give names to frequently used sequences of lexical items (not only values, but also program fragments). This compile-time substitution facility also allows a basic form of parametrization.

```
<simple-macro-directive> ::= #define <word> {<lex-item> | [ <lex-item> + ]}
```

The *word* is defined as a simple (parameterless) macro. Example: `#define zero? [0 = ]`. This shows that the sequence of lexical items need not be a proper construct of the language.

```
<parametrized-macro-directive> ::= #define <word>(<macro-arg> +)
      {[ <lex-item> + ] | ( <lex-item> + )}
```

The *word* is defined as a parametrized macro.

```
<macro-arg> ::= <word>
```

The parameters in the macro definition (*macro-arg*) must be *words*. All *words* must be different and they must not be equal to the names of built-in functions and operators (see section 5).

```
<macro-call> ::= <word> | <word>(<lex-item> +)
```

The *word* should have been defined as a macro name. At the point of usage, the *word* gets replaced by the *lex-item(s)* specified in the definition. If the macro is a parametrized one, every occurrence of a *macro-arg* among those *lex-items* will be replaced by the corresponding *lex-item* from the *macro-call*. In this case, the number of *lex-items* supplied in the *macro-call* should be equal to the number of *macro-args* in the definition; if the definition was a sequence enclosed in `( )`, the result is also enclosed in `( )`; if the sequence was enclosed in `[ ]`, these delimiters are removed.[†]

```
<lex-item> ::= "any lexical item of the Red/System language"
```

Lex-items that are delimiters occurring normally in pairs (`[ ]` `{ }` `( )` `" "`) can only be specified when properly paired.

```
<enumeration-directive> ::= #enum <word> [ {<label> | {<label>:} + <integer>} + ]
```

Enumeration-directives serve to define names for integer values, and synonyms for `integer!`.

The *word* following `#enum` is defined in the current scope as an *enumeration-type-name* that is treated as synonym with `integer!`. This may serve documentation purposes.

```
<label> ::= <word>
```

The *words* occurring as *labels* in the *enumeration-directive* are defined in the current scope as *enumeration-value-names* which may be used as integers. The have the same constraints as *enumeration-type-names*. The values allocated start at 0 for the first one and increase by 1 for every subsequent one, except if the *label* is followed by `:`, in which case the following *integer* specified is taken as the value; subsequent *labels* without `:` are again given values increasing by 1 etc. More than one *label* with the same integer value is allowed, and the specified integer values need not be in increasing order.

[†] Since the replacements described above are done before syntactic and semantic analysis, caution is needed when using macros. E.g. with the above example, the following program will not compile correctly:

```
#define zero? [0 = ] if zero? a + b [print "yes"]
```

since the expression `0 = a + b` will be analyzed as `(0 = a) + b` and a `logic!` value cannot be added to an `integer!` one. In other words: macros are not functions!

```
<interface-directive> ::= <import-directive> | <syscall-directive>
```

Interface-directives serve to define the interface to external library functions (*import-directive*) or internal system functions (*syscall-directive*).

```
<import-directive> ::= #import [ <library-functions>+ ]

<library-functions> ::=
      <c-string> {stdcall | cdecl} [ {<word>: <c-string> <ext-spec-block>}+ ]
```

The first *C-string* must identify a valid library file (`.dll` or equivalent). Each of the *words* will be defined as a library function, with calling convention `stdcall` or `cdecl`, whose body is identified within the library file by the following *C-string*, and with the corresponding *ext-spec-block*.

```
<syscall-directive> ::= #syscall [ {<word>: <integer> <ext-spec-block>}+ ]
```

Each of the *words* will be defined as a system-call function whose body is identified by the following *integer*, and with the corresponding *ext-spec-block*.

2. definitions

```
<definition> ::= <function-definition> | <context-definition> | <alias-definition>
```

Definitions serve to introduce names for functions and contexts (namespaces) as well as aliases for *struct-types* and *func-types*.

```
<function-definition> ::= <word>: {func | function} <spec-block> <body-block>
```

The *word* will be defined as a function with the indicated specification (*spec-block*) and body (*body-block*). It should not have been previously (w.r.t. compilation order) defined in the same scope, nor as a function in any scope.

```
<spec-block> ::= [ {[ <func-attrs> ]}° <formal-arg>* <return-spec>°
                        {/local <local-var>+}° ]

<func-attrs> ::= infix
            | stdcall | cdecl {variadic | typed}°| {variadic | typed} cdecl°
```

The usage of the attribute `infix` is discussed in section 4.

An attribute value of `cdecl` changes the function's calling convention to C convention. This allows to safely pass a Red/System function as argument to imported C functions. The default calling convention is `stdcall`.

The attributes `variadic` and `typed` each require that the function have exactly two *formal-args* (see next), one of type `integer!` representing the number of actual arguments and one representing a list of (pointers to) the actual arguments themselves, which are supplied at the point of *function-call* by a block of values. For a "variadic function", this second argument is of type `pointer![integer!]`, thus allowing for arguments of any type, while for a "typed function", the second argument is of type `struct![value [integer!] type [integer!]]`, allowing for transmission of type information with each actual argument, in the form of a type number (see Annex 3).

```
<formal-arg> ::= <word> [ <type> ] <c-string>°
```

The *word* is specified as a formal argument having the indicated *type*. It should not be the same as another formal argument of this function. The optional *C-string* is for documentation purposes only; it has no run-time effect.

```
<return-spec> ::= return: [ <type> ]
```

The *type* following `return:` is that of the function's return value.

```
<local-var> ::= <word> {[ <type> ]}°
```

The *word* is specified as a local variable, possibly with its *type*. It should not be the same as another local variable or formal argument of this function.

```
<ext-spec-block> ::= [ <ext-formal-arg>* <return-spec>° ]
```

An *ext-spec-block* is like a *spec-block*, but for library or operating system functions.

```
<ext-formal-arg> ::= <word> [ <ext-type> ] <c-string>°

<ext-type> ::= <type> | <func-type> | <func-alias>
```

```
<func-type> ::= function! [ <spec-block> ]
```

External functions (i.e. library functions and system-call functions) can take (callback) functions as arguments. In this case, the formal argument has a *func-type* instead of a normal *type*, and the actual argument must be an expression whose type is compatible (see section 8) with that *func-type*, e.g. a *get-word* where the *qualified-word* has been defined as a function, or the result of a *type-cast* where the *ext-type* is a *func-type*. The value `null`, meaning no function, is also admissible as an actual argument.

```
<body-block> ::=
      [ {<directive> | <definition> | <statement> | <expression> | <comment>}* ]
```

An *expression* occurring directly within the *body-block* of a *function-definition* will be evaluated but its value will be discarded, except if it is the last such *expression* evaluated in the *body-block*, and the function defined has a *return-spec*, in which case the value will be yielded as the result of the function, unless the result is determined in a subsequent *return-statement*. Indeed the evaluation of the last *expression* may be followed by the execution of any number of non-*expression* elements.

```
<context-definition> ::= <word>: context <body-block>
```

The *word* will be defined in the current scope to name the context ("namespace") of all names defined in directives, definitions and statements directly occurring in the *body-block*. It must not have been defined in the same or a containing scope. Outside the *body-block* (but within the scope of the context name) the defined variables etc. can be referred to by "qualification", see section 5. Contexts may be nested.

```
<alias-definition> ::= <word>: alias {<struct-type> | <func-type>}
```

The *word* will be defined as a `struct![…]` type or a `function![…]` pseudo-type. The *word* should not have been previously (w.r.t. compilation order) defined in the same scope nor as an *alias* in any scope and it may not be one of the *base-type* names (see section 7). Conventionally, such *words* end in `!`.

The *struct-type* may have *fields* whose *type* is the *word* being defined, thus making self-referencing possible. A *word* defined as *func-type* may be used in *ext-formal-args*, in a `declare` construct (see section 4) and in *type-casts*.

3. names and scopes: summary

Red/System has static scoping of names. Names which have a scope are of the following kinds:

- global variable names (defined implicitly by occurring in an *assignment* at *program* level)
- formal argument names (defined in the *spec-block* of a function)
- local variable names (idem)
- type names (pre-defined, defined in an *alias-definition* or by an *enumeration-directive*)
- function names (defined in an *interface-directive* or a *function-definition*)
- context names (defined in a *context-definition*)
- enumeration value names (defined by an *enumeration-directive*)

The scope of a global variable name comprises the whole *program.* The scope of formal argument names and local variable names is the *body-block* of the function in the *spec-block* of which they have been defined. The scope of a pre-defined type name is the whole *program*. The scope of any other type name, and of any function name, context name and enumeration value name is the *body-block* in which it has been defined. In all cases, names cannot be used before (in compilation order) they have been defined, except as explained under *alias-definition* in section 2.

The scope of a name does not include those *body-blocks* (of functions and contexts) which are contained within it, and in which another definition of that name is given.

Type names must be unique throughout the *program*, but variables (both global and local, including formal arguments), functions, contexts and enumeration values with the same names may co-exist with them. This is possible because type names are only used in very specific constructs and no confusion can arise.

Functions names must be unique throughout the *program*.

4. statements

```
<statement> ::=
        <function-call>
      | <assignment>
      | <return-statement>
      | <conditional-statement>
      | <with-block>
      | <assertion>

<function-call> ::=
        <fixed-arguments-function-call>
      | <variable-arguments-function-call>
```

If the *function-call* is a *statement*, the function may or may not have a return value, which will be discarded.

```
<fixed-arguments-function-call> :: = <qualified-word> <expression> *
```

The *qualified-word* must be defined as a function, which should not have the attribute `variadic` or `typed`. The *expressions*, if any, must correspond in number and be compatible in type with the formal arguments of the function (see section 8 for the type compatibility rules). The function will be called with the evaluated *expressions* as actual arguments. For arguments of type `c-string!`, `pointer![…]` and `struct![…]` and of peudo-type `function![…]`, the address will be passed, for others the value.

```
<variable-arguments-function-call> ::=
      <qualified-word> {<expression> | [ <expression> + ]}
```

The *qualified-word* must be defined as a function, which should have the attribute `variadic` or `typed`. The *expression(s)* will be used to construct the two actual arguments corresponding to the formal arguments as described under *func-attrs* above.

```
<assignment> ::= <lh-side> <rh-expression>
<lh-side> ::= {<qualified-word>: | <path>:}
```

An assignment associates the value of its *rh-expression* with a *qualified-word* or a *path*.

If the *assignment* is a direct element of a *program* and the *lh-side* is a *word*, and this is the lexically first occurrence of an *assignment* with this *word*, the *word* will be registered as a global variable having the type of the *rh-expression* (implicit definition). The type will be used for compatibility checks in subsequent (w.r.t compilation order) *assignments* involving this *word*. If the *assignment* is directly within the *body-block* of a *function-definition*, and the *lh-side* is a *word* which is not a global variable, it should have been explicitly defined as a formal argument (with its type) or a local variable (possibly with its type) in the corresponding *spec-block*. For local variables without a type, the lexically first occurrence of an *assignment* as a direct element of the *body-block* will serve to register the type. If the *assignment* is directly within the *body-block* of a *context-definition* and the *lh-side* is a *word*, it will be registered as a variable within the named context, having the type of the *rh-expression*.

For an *assignment* where the *lh-side* is a *qualified-word* which is not a *word*, or a *path*, no implicit definition is made and no type inference will occur. The *qualified-word* should have been defined, with its type, in a – possibly nested – context within the current scope or any containing scope. The first *word* of the *path* should have been defined, with its type, in the current or any containing scope. The type of the value resulting from the *rh-expression* should be compatible with that of the *lh-side* (see section 8 for detailed compatibility rules).

```
<rh-expression> ::=
        <expression>
      | declare {<pointer-type> | <struct-type> | <struct-alias> | <func-alias>}
```

Literal values of the types `pointer![…]` and `struct![…]`, and of the `function[…]` pseudo-type are introduced by `declare`. The actual value created is implementation dependent.

```
<return-statement> ::= exit | return <expression>
```

A *return-statement* may only occur within the *body-block* of a function. The `exit` statement causes return from the function without a return value. It is allowed only if the function has no return value specified. The `return` statement causes return with a value, namely that of the *expression*, which must have the type of the function's return value.

```
<conditional-statement> ::=
        if <cond-expression> <code-block>
      | either <cond-expression> <code-block> <code-block>
      | case [ {<cond-expression> <code-block>}+ ]
      | switch <expression> [ {<selectors> <code-block>+} {default <code-block>}° ]
      | until <cond-block>
      | while <cond-block> <code-block>
```

The semantics of these constructs follow closely those of the corresponding REBOL ones.

```
<cond-expression> ::= <expression>
```

The *expression* should yield a value of type `logic!`.

```
<cond-block> ::= [ {<statement> | <expression> | <comment>}+ ]
```

There should be at least one *expression* directly occurring within a *cond-block*. The value of the last such *expression* should be of type `logic!`; it is used for determining program flow in the `until` or `while` statement in which the *cond-block* occurs. No *statements* may follow this last *expression*.

```
<code-block> ::= [ {<statement> | <expression> | <comment>}+ ]
```

An *expression* occurring directly within a *code-block* will be evaluated but its value will be discarded.

```
<selectors> ::= <expression>+
```

Each *expression* must be of type `integer!` or `byte!` and all *expressions* occurring in the *selectors* must be of the same type which must be identical (but for synonyms) to that of the expression after `switch`.

```
<with-block> ::= with {<word> | [ <word>+ ]} <code-block>
```

The *word* should name a context. Within the *code-block*, the names defined in that context are available without needing qualification by path notation. If a list of *words* is specified and if the contexts named have one or more names defined in more than one of them, name resolution is done in the context that is defined last in compilation order.

```
<assertion> ::= assert <cond-expression>
```

This statement is intended to help in debugging. If the *cond-expression* is not true, a runtime error will be raised.

5. expressions

```
<expression> ::= <value> | <type-cast> | <short-circuit> | <size-enquiry>
      | <unary-operation> | <binary-operation>

<value> ::= <simple-value> | ( <expression> ) | <function-call>
```

If the *function-call* is a *value*, the function should have a return value, and should have the type of the return value specified in its *spec-block*. See further under *statement* in section 4.

```
<simple-value> ::= <literal> | <get-word> | <qualified-word> | <path>
```

A *qualified-word* (not having been defined as a function and not being a *literal-word*) or *path* used as *simple-value* should have been (defined and) initialized as a global or local variable, or defined as a formal argument. See also above, under *assignment*. For information on the allowed *literals* see section 9.

Note that a *path* is syntactically indistinguishable from a *qualified-word*. The distinction can only be made on the basis of the definition of the first *word* in each construct. As will be seen below, this is either a context name or a variable name.

```
<get-word> ::= :<qualified-word>
```

If the *qualified-word* has been defined as a function, the result is a value equal to the address of (the entry-point of) the function, which can only be used directly as argument in a *function-call* (where the function called is a library or system function), in an *assignment*, or in a *type-cast*. The type of this value is `function![<spec-block>]`, where `<spec-block>` is the *spec-block* of the function.
If the *qualified-word* has been defined as a variable of type `integer!`, `byte!`, `float!`, `float32!` or `float64!`, the result is a value of type `pointer![<pointed-type>]` where `<pointed-type>` is the type of the variable.

```
<qualified-word> ::= <word> | <qualifier><word>
```

If the *word* is preceded by a *qualifier*, it's definition is found in the (nested) context named by that *qualifier*.

```
<qualifier> ::= <context-name>/ | <qualifier><context-name>/
<context-name> ::= <word>
```

The *context-names* making up the *qualifier* are used to identify the (nested) context in which to find the definition of the *qualified-word*.

```
<path> ::= <path-head>/{<word> | <integer>}
```

Path expressions select components of types `c-string!`, `pointer![…]` or `struct![…]`. If the *path-head* yields a C-string value, both a *word* that is an integer variable or an *enumeration-value-name*, and an *integer* (i.e. integer literal) are allowed as selector. If the integer value is smaller than 1 or larger than the length of the string, the result is undefined, otherwise it selects a byte of the string (1-origin indexing), of type `byte!`. If the *path-head* yields a value of type `struct![…]`, only *words* equal to the *field* names are allowed; they select the corresponding *field* value. If the *path-head* yields a value of type `pointer![…]`, the *word* `value` (equivalent to 1), a *word* that is an integer variable or an *enumeration-value-name*, and an *integer* are allowed. If the integer value is not equal to 1, the result may be undefined, otherwise it selects the value (of type *pointed-type*) that the pointer points to.

```
<path-head> ::= <word> | <path>
```

The *word* should be a global variable or a formal argument or local variable of a function.

```
<type-cast> ::= as {<ext-type> | [ <ext-type> ]} <expression>
```

This signifies explicit type conversion. The brackets `[ ]` around the target type (*ext-type*) are optional, as shown. The *expression* must not be the literal `null` nor must it be itself a *type-cast*. The following combinations of *ext-type* and type of *expression* are allowed, with the indicated effect:

ext-type	type of *expression*	converted value
`logic!`	`integer!`, `byte!`	`0` or `#"^(00)"` -> `false` others->`true`
`integer!`, `byte!`	`logic!`	`false -> 0` or `#"^(00)"` `true -> 1` or `#"^(01)"`
`logic!`	`pointers!`	`null -> false`, others->`true`
`integer!`	`byte!`	same value `0..255`
`byte!`	`integer!`	truncated value (`mod 256`)
`integer!`	`float32!`	bit pattern kept
`float32!`	`integer!`	bit pattern kept
`integer!`	`any-pointer!`	address value
`any-pointer!`	`integer!`	address value
`function![…]`	`function![…]`	address value

Here `pointers! = c-string!` or `pointer![…]` or `struct![…]`
`any-pointer! = pointers!` or `function![…]`

All combinations not mentioned above are in error, except when the two types are equal in which case a warning will be given about the type-cast being unnecessary. For `function![…]` pseudo-types, "equality" means equality of the types of the formal arguments (not their names) and of the result, in addition to identity of the calling convention (attribute `cdecl` or `stdcall`); see also section 8.

```
<short-circuit> ::=
        either <cond-expression> <value-block> <value-block>
      | case [ {<cond-expression> <value-block>}+ ]
      | switch <expression> [ {<selectors> <value-block>+}
            {default <value-block>}° ]
      | {any | all} [ <cond-expression>+ ]
```

The *short-circuit* constructs with `either`, `case` and `switch` have the constraint that the resulting values of all *value-blocks* (see below) should have the same type. Otherwise they are identical in form to the corresponding *conditional-statements*. Thus if the constraint is not fulfilled and the *short-circuit*, as *expression*, occurs directly within a *program* or *body-block*, it will be considered a *statement*.
The expressions with `any` and `all` yield a logic value which is the logical disjunction (or) resp. conjunction (and) of all constituent *cond-expressions*.
In these *short-circuit* expressions, only those evaluations of the *(cond-)expressions* are done (in lexical order) that are necessary to determine the result.

```
<value-block> ::= [ {<statement> | <expression> | <comment>}+ ]
```

The last *expression* evaluated in a *value-block* is used as the resulting value in the corresponding alternative of the *short-circuit*. No *statements* may follow this expression.

```
<size-enquiry> ::= size? {<expression> | <base-type>}
```

A *size-enquiry* yields an implementation-dependent integer value, which is the number of bytes in memory occupied by the value of the *expression*, resp. by a value of (any type suggested by) the *base-type*. When the *expression* is of composite type (i.e. represents an address), the size applies to the value pointed to, not to the address itself. For *base-types* this is not the case.

```
<unary-operation> ::= not <expression>
```

The operand of the `not` operator should be of `integer!` or `logic!` type and the result has the same type. Applied to an integer value the result is the one's complement (`not 0 => -1`).

```
<binary-operation> ::= <expression> <infix-operator> <value>
```

Infix-operators are applied without precedence, from left to right. They are applied before *function-calls*, *unary-operators* and *type-casts*.

```
<infix-operator> ::= <math-op> | <bitwise-op> | <comparison-op> | <word>
```

The *word* should be defined as an infix function (i.e. one having *func-attrs* `[infix]`). This function should have two formal arguments (operands) and specify the type of its return value.

```
<math-op> ::= + | - | * | / | % | //
```

The signs `+ - * /` have their usual mathematical meaning. The sign `%` means remainder (result has the sign of the divider) and `//` means modulo (positive result).

```
<bitwise-op> ::= and | or | xor | << | >> | >>>
```

The operators `and or xor` operate as expected; the sign `<<` is left shift, `>>` and `>>>` are right shift (with and without sign extension).

```
<comparison-op> ::= = | <> | > | < | >= | <=
```

Also these signs have their usual mathematical meaning.

6. binary operations

The admissible operand types and the result type of the literal *infix-operators* are as follows:

operator	left operand	right operand	result
`+ -`	`poly!`	`poly!`	same as left operand
`* / % //`	`any-number!`	`any-number!`	same as left operand
`<< >> >>>`	`number!`	`number!`	same as left operand
`and or xor`	`number!`	`number!`	same as left operand
`= <>`	`any-type!`	`any-type!`	`logic!`
`> < >= <=`	`poly!`	`poly!`	`logic!`

Here

`number!` = `integer!` or `byte!`

`any-number!` = `number!` or `float!` or `float64!` or `float32!`

`poly!` = `any-number!` or `any-pointer!` (see also above, under *type-cast*)

`any-type!` = `poly!` or `logic!`

Infix operations on `any-pointer!` operands always use the machine address as operand value.

Except for the bitshift operators `<< >> >>>`, the two operands should have equal type. In addition, for the combinations `integer!/byte!` v.v. all *math-ops* are allowed. Also, with the operators `+` and `-`, values of type `any-pointer!` can be freely combined with each other and with `integer!` values; note that where the second operand is of type `integer!`, its value is scaled by the size (number of bytes occupied in memory) of the value that the pointer points to, as is the case in C.

Furthermore, the value `null` can be compared with any value of type `any-pointer!`.

7. types

```
<type> ::= <simple-type> | <composite-type> | <struct-alias>

<simple-type> ::= integer! | byte! | logic! | float! | float32!
      | float64! | <enumeration-type-name>
```

The type names `float!` and `float64!` are synonyms. The value ranges of these *simple-types* are: `integer!` – 32 bit signed integers, `byte!` – 8 bit unsigned integers, `logic!` – `true/false`, `float!` – IEEE-754 64 bit double precision floating point numbers, `float32!` – IEEE-754 32 bit single precision floating point numbers.

```
<enumeration-type-name> :: = <word>
```

The *word* should have been defined as an enumeration type name in an *enumeration-directive* (see section 1). It is treated as synonym with `integer!`.

```
<composite-type> ::= c-string! | <pointer-type> | <struct-type>
```

A value of type `c-string!` is a null-terminated sequence of bytes representing a string of characters from a machine- and OS-dependent character set.

```
<pointer-type> ::= pointer! [<pointed-type>]
```

A value of a *pointer-type* is a machine address interpreted as a pointer to a typed value.

```
<pointed-type> ::= integer! | byte! | float! | float32! | float64!
```

Further alternatives for *pointed-type* may be defined, e.g. `<enumeration-type-name>`.

```
<struct-type> ::= struct! [<struct-attr> ° <field> +]
```

A value of a *struct-type*, or "struct", represents aggregate data consisting of named and typed value *fields*.

```
<struct-attr> ::= align <integer> {big | little} °
```

Structure attributes (*struct-attr*) are not yet implemented.

```
<field> ::= <word> [ <type> ]
```

The *word* is specified as a named field having the indicated *type*.

```
<base-type> ::= <simple-type> | c-string! | <alias>
      | pointer! | struct! | function!
```

A *base-type* may be used as argument for the `size?` function, and where it is not a *simple-type*, `c-string!` or *alias*, it represents any *ext-type* starting with one of the indicated words.

```
<alias> ::= <struct-alias> | <func-alias>

<struct-alias> ::= <word>
```

The *word* should have been defined as a `struct![…]` type, in an *alias-definition*.

```
<func-alias> ::= <word>
```

The *word* should have been defined as a `function![…]` pseudo-type, in an *alias-definition*.

```
<ext-type> ::= <type> | <func-type> | <func-alias>
```

8. type compatibility rules

Type compatibility applies in the following cases: between the source type and target type of a *type-cast*, between the left and right operands of an *infix-operator*, between the left hand side and the right hand side of an *assignment*, and between each actual argument of a *function-call* and the corresponding formal argument of the function called.

The rules concerning *type-casts* are given in section 5. The case of a literal *infix-operator* is treated in section 6. The application of a user-defined function that has the attribute `infix` is treated like a normal *function-call* of two arguments, see immediately below.

In the case of an *assignment* or a *function-call*, the requirement is "equality" of the two types after resolution of aliases and synonyms. For *simple-types* and `c-string!` this means identity of the type names. For `struct![…]` types, equality means that corresponding field names and field types must be identical. For `function![…]` pseudo-types, the *spec-blocks* are compared for equality of the types of the formal arguments (not their names) and of the result, in addition to identity of the calling convention (attribute `cdecl` or `stdcall`).

The value `null` is compatible with left hand sides resp. formal arguments of `c-string!`, `pointer![…]` and `struct![…]` type and of `function![…]` pseudo-type.

9. literals and words

```
<literal> ::= <integer> | <byte> | <decimal> | <c-string> | <literal-word>

<integer> ::= "REBOL literal integer value"
```

The current compiler version (written in REBOL) accepts `'` signs in *integers*, even though these do not seem to be part of the specification. Integers in hex notation are pre-processed by the Red/System loader.

```
<byte> ::= "REBOL literal char value"

<decimal> ::= "REBOL literal decimal value"

<c-string> ::= "REBOL literal string value"

<literal-word> ::= true | false | null | <enumeration-value-name>
```

The *literal-words* `true` and `false` are of type `logic!`. The *literal-word* `null` is pre-defined as an address value `0` that is compatible with `c-string!` and any `pointer![…]` or `struct![…]` type, and `function![…]` pseudo-type.

```
<enumeration-value-name> :: = <word>
```

The *word* should have been defined as an enumeration value name in an *enumeration-directive* (see section 1). Its value is of type `integer!`.

```
<word> ::= "REBOL word, except reserved words, and with restrictions"
```

Names of (global and local) variables, formal arguments, functions and types, as well as enumeration types and values, aliases and context-names are *words* which are a subset of the REBOL values of type `word`!: they are composed of characters in the range `21h` to `7Eh` with the exception of the following set:

```
[ ] { } " ( ) / \ @ # $ % ^ , : ; < >
```

and they cannot begin with a digit `0-9` or a single quote `'`. Thus in addition to the alphanumeric characters, the following are allowed: `! & ' * + - . = ? _ ` | ~`, but of these `'` not at the beginning. Names are case-insensitive. Conventionally, type names end in `!`

Annex 1: reserved and predefined words

The following words have special significance in Red/System:

keywords and operator symbols

```
alias func function declare
if either case switch
until while context with
assert exit return comment
as not size? any all
and or xor
= <> > < >= <=
+ - * / // % ///
<< >> >>> -**
```

literal words

```
true false null
```

predefined functions

```
pop push
```

base types

```
integer! byte! logic! float! float32! float64!
c-string! pointer! struct! function!
```

attribute names

```
stdcall cdecl infix
align big little
```

field name

```
value
```

special structure name

```
system
```

The reserved words are shown in **bold** in the above list. An attempt to redefine them will give a compilation error. The *base-types* can be redefined as variables or functions, but not as type names. The word `&` is reserved for future use, but this reservation is not enforced by the current compiler.

The words `///` and `-**` are temporary internal replacements for `%` and `>>>`, which are not accepted by the REBOL loader and cannot be used in the version of the compiler written in REBOL.

Note that the current compiler for Red/System uses a number of standard "include files" with macro definitions that effectively predefine a large number of words in addition to those listed above, such as `byte-ptr!` (for `pointer! [byte!]`) etc. Since these definitions may be changed without affecting the language as described, and since their meaning may vary with the target operating system and machine architecture, they are not listed here.

Annex 2: the system structure

Red/System offers direct access to a number of objects through the predefined structure `system`. This is defined as follows:

```
system: declare struct! [                   ;-- store runtime accessible system values
        args-count      [integer!]          ;-- command-line arguments count (do not move)
        args-list       [str-array!]        ;-- command-line arguments array pointer (do not move)
        env-vars        [str-array!]        ;-- environment variables array pointer
                                            ;   (always null for Windows)
        stack           [__stack!]          ;-- stack virtual access
        pc              [pointer! [byte!]]  ;-- CPU program counter value
;       cpu             [__cpu-struct!]     ;-- reserved for later use
        fpu             [__fpu-struct!]     ;-- FPU settings
        alias           [integer!]          ;-- aliases ID virtual access
        words           [integer!]          ;-- global context accessor (dummy type)
]
__stack!: alias struct! [
        top             [pointer! [integer!]]]
        frame           [pointer! [integer!]]]
]
__fpu-struct!: alias struct! [              ;-- Intel x87 case
        type            [integer!]
        option          [x87-option!]
        mask            [x87-mask!]
        control-word    [integer!]          ;-- direct access to whole control word
        epsilon         [integer!]          ;-- Ulp threshold for almost-equal op (not used yet)
        update          [integer!]          ;-- action simulated using a read-only member
]
x87-mask!: alias struct! [                  ;-- x87 exception mask (true => disable exception)
        precision       [logic!]
        underflow       [logic!]
        overflow        [logic!]
        zero-divide     [logic!]
        denormal        [logic!]
        invalid-op      [logic!]
]
x87-option!: alias struct! [
        rounding        [integer!]
        precision       [integer!]
]
str-array!: alias struct! [
        item [c-string!]
]
```

Accesses (both get and set) to the fields `stack`, `pc` and `fpu` of this structure are caught by the compiler/emitter and translated specially.

The field `system/stack` represents the run-time stack that is used for argument transmission and other purposes (saving of intermediate results). The paths `system/stack/top` and `system/stack/frame` can be used to get and set the addresses of the stack top and stack frame respectively, as a value of type `pointer![integer!]`. Furthermore there is defined a built-in function `pop` of no arguments, which yields the value of `system/stack/top` as an `integer!` and in addition pops the stack, and a function `push` of one argument, of any type, which pushes the value of its argument on the stack.

The field `system/pc` represents the program counter as a byte address (get only).

The field `system/fpu` gives access to properties of the floating point unit (if present). For illustration the Intel x87 architecture case is shown.

The field `system/alias` gives access to the internal type numbers of type names defined through an *alias-definition*. Conventionally, these type numbers are 1000 + the ordinal number of the alias name in the internal hash table of alias definitions.

The field `system/words` gives access to the global definitions of names that may have been redefined locally.

Annex 3: type numbers

```
logic!          1
integer!        2
byte!           3
float32!        4
float!          5
c-string!       6
byte-ptr!       7
int-ptr!        8
function!       9
struct!         1000
aliases         1001 …
```