Superpowered game development.

Language Syntax

version 3.0.4576 beta

Live/current version at http://SkookumScript.com/docs/

April 11, 2017

Better coding through mad science.

Combined syntactical and lexical rules for SkookumScript in modified Extended Backus-Naur Form (EBNF). Production rules in *italics*. Terminals **`coloured and in bold`** and literal strings ‘**`quoted`**’. Optional groups: []. Repeating groups of zero or more: {}. Repeating groups of n or more: $\{\}^{n+}$. Mandatory groups: (). Alternatives (exclusive or): |. Disjunction (inclusive or): V.

File Names and Bodies:

```
method-filename[1]    =  method-name ‘()’ [‘C’] ‘.sk’
method-file[2]        =  ws {annotation wsr} parameters [ws code-block] ws
coroutine-filename    =  coroutine-name ‘()’ ‘.sk’
coroutine-file[3]     =  ws {annotation wsr} parameter-list [ws code-block] ws
data-filename[4]      =  ‘!Data’ [‘C’] ‘.sk’
data-file             =  ws [data-definition {wsr data-definition} ws]
data-definition[5]    =  {annotation wsr} [class-desc wsr] ‘!’ data-name
annotation[6]         =  ‘&’ instance-name
object-id-filename[7] =  class-name [‘-’ {printable}]  ‘.sk’ ‘-’ | ‘~’ ‘ids’
object-id-file[8]     =  {ws symbol-literal | raw-object-id} ws
raw-object-id[9]      =  {printable}^1-255 end-of-line
```

Expressions:

```
expression            =  literal | identifier | flow-control | primitive | invocation
```

Literals:

```
literal               =  boolean-literal | integer-literal | real-literal | string-literal | symbol-literal
                         | char-literal | list-literal | closure
boolean-literal       =  ‘true’ | ‘false’
integer-literal[10]   =  [‘-’] digits-lead [‘r’ big-digit {[number-separator] big-digit}]
real-literal[11]      =  [‘-’] digits-lead V (‘.’ digits-tail) [real-exponent]
real-exponent         =  ‘E’ | ‘e’ [‘-’] digits-lead
digits-lead[12]       =  ‘0’ | (non-zero-digit {[‘_’] digit})
digits-tail           =  digit {[‘_’] digit})
string-literal        =  simple-string {ws ‘+’ ws simple-string}
simple-string         =  ‘"’ {character} ‘"’
symbol-literal        =  ‘'’ {character}^0-255 ‘'’
char-literal          =  ‘`’ character
list-literal[13]      =  [(list-class constructor-name invocation-args) | class-name]
                         ‘{’ ws [expression {ws [‘,’ ws] expression} ws] ‘}’
```

[1] If optional ‘**?**’ is used in query/predicate method name, use ‘**-Q**’ as a substitute since question mark not valid in filename.

[2] Only immediate calls are permissible in the code block. If *code-block* is absent, it is defined in C++.

[3] If *code-block* is absent, it is defined in C++.

[4] A file name appended with ‘**C**’ indicates that the file describes class members rather than instance members.

[5] *class-desc* is compiler hint for expected type of member variable. If class omitted, **`Object`** inferred or **Boolean** if *data-name* ends with ‘**?**’. If *data-name* ends with ‘**?**’ and *class-desc* is specified it must be **Boolean**.

[6] The context / file where an *annotation* is placed limits which values are valid.

[7] Starts with the object id class name then optional source/origin tag (assuming a valid file title) – for example: Trigger-WorldEditor, Trigger-JoeDeveloper, Trigger-Extra, Trigger-Working, etc. A dash ‘**-**’ in the file extension indicates an id file that is a compiler dependency and a tilde ‘**~**’ in the file extension indicates that is not a compiler dependency

[8] Note: if *symbol-literal* used for id then leading whitespace, escape characters and empty symbol (**`''`**) can be used.

[9] Must have at least 1 character and may not have leading whitespace (*ws*), single quote (‘**'**’) nor *end-of-line* character.

[10] ‘**r**’ indicates *digits-lead* is (r)adix/base from 1 to 36 - default 10 (decimal) if omitted. Ex: **`2r`** binary & **`16r`** hex. Valid *big-digit*(s) vary by the radix used. See *math-operator* footnote on how to differentiate subtract from negative *integer-literal*.

[11] Can use just *digits-lead* if **`Real`** type can be inferred from context otherwise the *digits-tail* fractional or *real-exponent* part is needed. See *math-operator* footnote on how to differentiate subtract from negative *real-literal*.

[12] ‘**_**’ visually separates parts of the number and ignored by the compiler.

[13] Item type determined via optional *list-class* constructor or specified class. If neither supplied, then item type inferred using initial items, if no items then **`Object`** used.

```
closure[1]             =   (‘^’ [‘_’ ws] [expression ws]) ∨ (parameters ws) code-block
```

Identifiers:

```
identifier[2]          =   variable-identifier | reserved-identifier | class-name | object-id
variable-identifier[3] =   variable-name | ([expression ws ‘.’ ws] data-name)
variable-name          =   name-predicate
data-name[4]           =   ‘@’ | ‘@@’ variable-name
reserved-identifier    =   ‘nil’ | ‘this’ | ‘this_class’ | ‘this_code’ | ‘this_mind’
object-id[5]           =   [class-name] ‘@’ [‘?’ | ‘#’] symbol-literal
invoke-name            =   method-name | coroutine-name
method-name[6]         =   name-predicate | constructor-name | destructor-name | class-name
name-predicate[7]      =   instance-name [‘?’]
constructor-name       =   ‘!’ [instance-name]
destructor-name[8]     =   ‘!!’
coroutine-name         =   ‘_’ instance-name
instance-name          =   lowercase {alphanumeric}
class-name             =   uppercase {alphanumeric}
```

Flow Control:

```
flow-control           =   code-block | conditional | case | when | unless | | loop | loop-exit | concurrent
                           | class-cast | class-conversion
code-block             =   ‘[’ ws [expression {wsr expression} ws] ‘]’
conditional            =   ‘if’ {ws expression ws code-block}1+ [ws else-block]
case                   =   ‘case’ ws expression {ws expression ws code-block}1+ [ws else-block]
else-block             =   ‘else’ ws code-block
when                   =   expression ws ‘when’ ws expression
unless                 =   expression ws ‘unless’ ws expression
loop[9]                =   ‘loop’ [ws instance-name] ws code-block
loop-exit[10]          =   ‘exit’ [ws instance-name]
concurrent             =   sync | race | branch | divert
sync[11]               =   ‘sync’ ws code-block
race[12]               =   ‘race’ ws code-block
branch[13]             =   ‘branch’ ws expression
change[1]              =   ‘change’ ws expression ws expression
```

[1] [AKA code block/anonymous function/lambda expression] Optional ‘^’, *parameters* or both must be provided (unless used in *closure-tail-args* where both optional). Optional *expression* (may not be *code-block*, *closure* or *routine-identifier*) captured and used as receiver/this for *code-block* - if omitted this inferred. Optional ‘_’ indicates it is durational (like coroutine) - if not present durational/immediate inferred via *code-block*. Parameter types, return type, scope, whether surrounding this or temporary/parameter variables are used and captured may all be inferred if omitted.

[2] Scoping not necessary - instance names may not be overridden and classes and implicit identifiers effectively have global scope.

[3] Optional *expression* can be used to access data member from an object - if omitted, this is inferred.

[4] ‘@’ indicates instance data member and ‘@@’ indicates class instance data member.

[5] If *class-name* absent, Actor inferred or desired type if known. If optional ‘?’ present and object not found at runtime then result is nil else assertion error occurs. Optional ‘#’ indicates no lookup - just return name identifier validated by class type.

[6] A method using *class-name* allows explicit conversion similar to *class-conversion* except that the method is always called.

[7] Optional ‘?’ used as convention to indicate predicate variable or method of return type Boolean (true or false).

[8] Destructor calls are only valid in the scope of another destructor’s code block.

[9] The optional *instance-name* names the loop for specific reference by a *loop-exit* which is useful for nested loops.

[10] A *loop-exit* is valid only in the code block scope of the loop that it references.

[11] 2+ durational expressions run concurrently and next *expression* executed when *all* expressions returned (result nil, return args bound in order of expression completion).

[12] 2+ durational expressions run concurrently and next *expression* executed when *fastest* expression returns (result nil, return args of fastest expression bound) and other expressions are *aborted*.

[13] Durational expression run concurrently with surrounding context and the next *expression* executed immediately (result InvokedCoroutine). *expression* is essentially a closure with captured temporary variables to ensure temporal scope safety. Any return arguments will be bound to the captured variables.

Invocations:

```
invocation          =  invoke-call | invoke-cascade | apply-operator | invoke-operator | index-operator
                       | instantiation
invoke-call[2]      =  ([expression ws '.' ws] invoke-selector) | operator-call
invoke-cascade      =  expression ws '.' ws '[' {ws invoke-selector | operator-selector}2+ ws ']'
apply-operator[3]   =  expression ws '%' | '%>' invoke-selector
invoke-operator[4]  =  expression bracketed-args
index-operator[5]   =  expression '{' ws expression ws '}' [ws binding]
instantiation[6]    =  class-instance | expression '!' [instance-name] invocation-args
invoke-selector     =  [scope] invoke-name invocation-args
scope               =  class-name '@'
operator-call[7]    =  (prefix-operator ws expression) | (expression ws operator-selector)
operator-selector   =  postfix-operator | (binary-operator ws expression)
prefix-operator[8]  =  'not' | '-'
binary-operator     =  math-operator | compare-op | logical-operator | ':='
math-operator[9]    =  '+' | '+=' | '-' | '-=' | '*' | '*=' | '/' | '/='
compare-op          =  '=' | '~=' | '>' | '>=' | '<' | '<='
logical-operator[10] = 'and' | 'or' | 'xor' | 'nand' | 'nor' | 'nxor'
postfix-operator    =  '++' | '--'
invocation-args[11] =  [bracketed-args] | closure-tail-args
bracketed-args      =  '(' ws [send-args ws] [';' ws return-args ws] ')'
closure-tail-args[12] = ws send-args ws closure [ws ';' ws return-args]
send-args           =  [argument] {ws [',' ws] [argument]}
return-args         =  [return-arg] {ws [',' ws] [return-arg]}
argument            =  [named-spec ws] expression
return-arg[13]      =  [named-spec ws] variable-identifier | define-temporary
named-spec[14]      =  variable-name ws ':'
```

[1] Rather than inheriting the caller's updater **Mind** object, durational expressions in the second expression are updated by the mind object specified by the first expression.

[2] If an *invoke-call*'s optional *expression* (the receiver) is omitted, '**this.**' is implicitly inferred.

[3] If **List**, each item (or none if empty) sent call - coroutines called using **%** - **sync**, **%>** - **race** respectively and returns itself (the list). If non-list it executes like a normal invoke call - i.e. '**%**' is synonymous to '**.**' except that if **nil** the call is ignored, then the normal result or **nil** respectively is returned.

[4] Akin to **expr.invoke(…)** or **expr._invoke(…)** depending if *expression* immediate or durational - *and* if enough context is available the arguments are compile-time type-checked plus adding any default arguments.

[5] Gets item (or sets item if *binding* present) at specified index object. Syntactic sugar for **at()** or **at_set()**.

[6] *expression* used rather than *class-instance* provides lots of syntactic sugar: **expr!ctor()** is alias for **ExprClass!ctor(expr)** - ex: **num!copy** equals **Integer!copy(num)**; brackets are optional for *invocation-args* if it can have just the first argument; a constructor-name of **!** is an alias for **!copy** - ex: **num!** equals **Integer!copy(num)**; and if **expr!ident** does not match a constructor it will try **ExprClass!copy(expr).ident** - ex: **str!uppercase** equals **String!copy(str).uppercase**.

[7] Every operator has a named equivalent. For example **:=** and **assign()**. Operators do *not* have special order of precedence - any order other than left to right must be indicated by using code block brackets (**[** and **]**).

[8] See math-operator footnote about subtract on how to differentiate from a negation '**-**' prefix operator.

[9] In order to be recognized as single subtract '**-**' expression and not an *expression* followed by a second *expression* starting with a minus sign, the minus symbol '**-**' must either have whitespace following it or no whitespace on either side.

[10] Like other identifiers - whitespace is required when next to other identifier characters.

[11] *bracketed-args* may be omitted if the invocation can have zero arguments

[12] Routines with last send parameter as mandatory closure may omit brackets '**()**' and closure arguments may be simple *code-block* (omitting '**^**' and parameters and inferring from parameter). Default arguments indicated via comma '**,**' separators.

[13] If a temporary is defined in the *return-arg*, it has scope for the entire surrounding code block.

[14] Used at end of argument list and only followed by other named arguments. Use compatible **List** object for group argument. Named arguments evaluated in parameter index order regardless of call order since defaults may reference earlier parameters.

Primitives:

```
primitive             =   create-temporary | bind | class-cast | class-conversion
create-temporary      =   define-temporary [ws binding]
define-temporary      =   '!' ws variable-name
bind[1]               =   variable-identifier ws binding
binding[2]            =   ':' ws expression
class-cast[3]         =   expression ws '<>' [class-desc]
class-conversion[4]   =   expression ws '>>' [class-name]
```

Parameters:

```
parameters[5]         =   parameter-list [ws class-desc]
parameter-list        =   '(' ws [send-params ws] [';' ws return-params ws] ')'
send-params           =   parameter {ws [',' ws] parameter}
return-params         =   param-specifier {ws [',' ws] param-specifier }
parameter             =   unary-param | group-param
unary-param[6]        =   param-specifier [ws binding]
param-specifier[7]    =   [class-desc wsr] variable-name
group-param           =   group-specifier
group-specifier[8]    =   '{' ws [class-desc {wsr class-desc} ws] '}' ws instance-name
```

Class Descriptors:

```
class-desc            =   class-unary | class-union
class-unary           =   class-instance | meta-class
class-instance        =   class-name | list-class | invoke-class
meta-class            =   '<' class-name '>'
class-union[9]        =   '<' class-unary {'|' class-unary}1+ '>'
invoke-class[10]      =   ['_' | '+'] parameters
list-class[11]        =   List '{' ws [class-desc ws] '}'
```

[1] Compiler gives warning if *bind* used in *code-block* of a *closure* since it will be binding to captured variable not original variable in surrounding context. May not be used as an argument.

[2] [Stylisticly prefer no *ws* prior to ':' - though not enforcing it via compiler.]

[3] Compiler *hint* that expression evaluates to specified class - otherwise error. *class-desc* optional if desired type can be inferred. If *expression* is *variable-identifier* then parser updates type context. [Debug: runtime ensures class specified is received.]

[4] Explicit conversion to specified class. *class-name* optional if desired type inferable. Ex: `42>>String` calls convert method `Integer@String()` i.e. `42.String()` - whereas `"hello">>String` generates no extra code and is equivalent to `"hello"`.

[5] Optional *class-desc* is return class - if type not specified `Object` is inferred (or **Boolean** type for predicates or **Auto_** type for closures) for nested parameters / code blocks and `InvokedCoroutine` is inferred for coroutine parameters.

[6] The optional *binding* indicates the parameter has a default argument (i.e. supplied *expression*) when argument is omitted.

[7] If optional *class-desc* is omitted `Object` is inferred or **Auto_** for closures or **Boolean** if *variable-name* ends with '?'. If *variable-name* ends with '?' and *class-desc* is specified it must be **Boolean**.

[8] `Object` inferred if no classes specified. Class of resulting list bound to *instance-name* is class union of all classes specified.

[9] Indicates that the class is any one of the classes specified and which in particular is not known at compile time.

[10] '_' indicates durational (like coroutine), '+' indicates durational/immediate and lack of either indicates immediate (like method). Class '`Closure`' matches any closure interface. Identifiers and defaults used for parameterless closure arguments.

[11] `List` is any `List` derived class. If *class-desc* in item class descriptor is omitted, `Object` is inferred when used as a type or the item type is deduced when used with a *list-literal*. A *list-class* of any item type can be passed to a simple untyped `List` class.

Whitespace:

wsr[1]	=	{*whitespace*}$^{1+}$
ws	=	{*whitespace*}
whitespace	=	*whitespace-char* | *comment*
whitespace-char	=	‘ ’ | formfeed | newline | carriage-return | horiz-tab | vert-tab
end-of-line	=	newline | carriage-return | end-of-file
comment	=	*single-comment* | *multi-comment*
single-comment	=	‘//’ {printable} *end-of-line*
multi-comment	=	‘/*’ {printable} [*multi-comment* {printable}] ‘*/’

Characters and Digits:

character	=	*escape-sequence* | printable
escape-sequence[2]	=	‘\’ *integer-literal* | printable
alphanumeric	=	*alphabetic* | *digit* | ‘_’
alphabetic	=	*uppercase* | *lowercase*
lowercase	=	‘a’ | … | ‘z’
uppercase	=	‘A’ | … | ‘Z’
digits	=	‘0’ | (*non-zero-digit* {*digit*})
digit	=	‘0’ | *non-zero-digit*
non-zero-digit	=	‘1’ | ‘2’ | ‘3’ | ‘4’ | ‘5’ | ‘6’ | ‘7’ | ‘8’ | ‘9’
big-digit	=	*digit* | *alphabetic*

[1] *wsr* is an abbreviation for (w)hite (s)pace (r)equired.

[2] Special escape characters: ‘n’ - newline, ‘t’ - tab, ‘v’ - vertical tab, ‘b’ - backspace, ‘r’ - carriage return, ‘f’ - formfeed, and ‘a’ - alert. All other characters resolve to the same character including ‘\’, ‘”’, and ‘’’.