Mightier Game Development

Amazing gameplay, level design & AI without melting your brain

Language Syntax

ver. 3!27 Proposed Syntax

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

Conan Reis

conan@AgogLabs.com

October 27, 2014

Fun games lovingly crafted by mad scientists

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.

Highlight colouring key: in progress, planned, under consideration.

File Names and Bodies:

method-filename[1]	=	*method-name* ‘**()**’ [‘**C**’] ‘**.skoo**’
method-file[2]	=	*ws parameters* [*ws code-block*] *ws*
coroutine-filename	=	*coroutine-name* ‘**()**’ [‘**C**’] ‘**.skoo**’
coroutine-file[3]	=	*ws parameter-list* [*ws code-block*] *ws*
data-filename[4]	=	‘**!Data**’ [‘**C**’] ‘**.skoo**’
data-file	=	*ws* [*data-definition* {*wsr data-definition*} *ws*]
data-definition[5]	=	[*class-desc wsr*] ‘**!**’ *data-name* [*ws binding*]
object-id-filename[6]	=	*class-name* [‘**-**’ {**printable**}] ‘**.skoo**’ ‘**-**’ | ‘**~**’ ‘**ids**’
object-id-file[7]	=	{*ws symbol-literal* | *raw-object-id*} *ws*
raw-object-id[8]	=	{**printable**}$^{1\text{-}255}$ *end-of-line*
enumeration-file	=	*ws* {*enum-definition ws*}
enum-definition	=	*enumeration-name ws* [‘**:**’ *ws enum-class ws*]
		‘**[**’ *ws* [*enumerator-defn* {*wsr enumerator-defn*} *ws*] ‘**]**’
enumeration-name	=	‘**#**’ *alphabetic* {*alphanumeric*}
enumerator-defn[9]	=	*instance-name* [*ws* ‘**:**’ *ws integer-literal*]
flagset-file	=	*ws* {*flagset-definition ws*}
flagset-definition	=	*flagset-name ws* [‘**:**’ *ws flagset-class ws*]
		‘**[**’ *ws* [*flag-definition* {*wsr flag-definition*} *ws*] ‘**]**’
flag-definition[10]	=	*flag-name* [*ws* ‘**:**’ *ws flag-operand*]
flag-name	=	*instance-name*
flag-operand[11]	=	*digits* | *flag-name* | *flag-op* | *flag-group*
flag-group[12]	=	‘**[**’ *ws flag-op ws* ‘**]**’
flag-op	=	*flag-operand ws flag-operator ws flag-operand*
flag-operator	=	*logical-operator* | ‘**-**’

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. [Combine data files into one – add a keyword to separate instance and class and change name to “Class”.]

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**. Optional binding part is default initialization. If default binding omitted, member must be bound to appropriate object before exiting constructor (or class constructor). [If default binding present its result class could infer class. Default bindings especially useful if classes used as “property sheets”.]

6 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

7 Note: if *symbol-literal* used for id then leading whitespace, escape characters and empty symbol can be used.

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

9 Assigning an enumerator to an integer is discouraged though it is often handy to mirror underlying C++.

10 If optional bit digit assignment used it is a ‘persistent flag’. A flag assigned to another single flag is an ‘aliased flag’. A flag assigned to a combination of flags using operations is a ‘flag group’. If optional assignment is omitted, an unassigned bit is used.

11 Valid digits range from 0 to 31 (i.e. 32-bits).

12 [*flag-group* could enclose any *flag-operand*, but grouping only has an effect around a *flag-op*, so this helps keep things tidy.]

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 | range-literal | closure-routine | map-literal
                       | enumerator | flagset-literal
boolean-literal    =   'true' | 'false'
integer-literal[1] =   ['-'] digits-lead ['r' big-digit {[number-separator] big-digit}]
real-literal[2]    =   ['-'] digits-lead V ('.' digits-tail) [real-exponent]
real-exponent      =   'E' | 'e' ['-'] digits-lead
digits-lead        =   '0' | (non-zero-digit {[number-separator] digit})
digits-tail        =   digit {[number-separator] digit})
number-separator[3] =  '_'
string-literal     =   escaped-string | raw-string [ws '+' ws string-literal]
escaped-string[4]  =   '"' {character | ('\' [bracketed-args] code-block)} '"'
raw-string[5]      =   'R' ['-' ['-']] '"' {printable}^0-16 '(' {printable} ')' {printable}^0-16 '"'
symbol-literal     =   ''' {character}^0-255 '''
char-literal       =   '`' character
list-literal[6]    =   [(list-class constructor-name invocation-args) | class-desc]
                       '{' ws [expression {ws [',' ws] expression} ws] '}'
closure[7]         =   ('^' ['^'] ['_' ws] [expression ws]) V (parameters ws) code-block
range-literal[8]   =   [expression] '..' [['.'] expression] | ('#' expression)
closure-routine[9] =   '^' routine-identifier
map-literal[10]    =   [(map-class constructor-name invocation-args) | (class-desc ':' ws [class-desc ws])]
                       '{' ws (key-value {ws [',' ws] key-value}) | ':' ws '}'
key-value          =   expression ws binding
enumerator[11]     =   (enum-class '.') | '#' instance-name
flagset-literal    =   (flagset-class '.') | '##' (flag-name | 'all' | 'none')
```

1 '**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*.

2 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*.

3 Visually separates parts of the number and ignored by the compiler. [Consider adding ''' since it will be used by C++.]

4 Raw string using syntax similar to C++11. Escaped *code-block* indicates use of string interpolation with resulting object having **String()** conversion method called on it. If optional *bracket-args* present it is used as argument(s) to **String()** call.

5 Optional single dash '-' indicates initial & ending whitespace removed from string. Optional double dash '--' removes initial and ending whitespace and indentation of first line from all lines. Optional character sequence prior to opening parenthesis '(' used to make unique delimiter pair that must be matched with the closing character sequence following closing parenthesis ')'.

6 Item type determined via optional *list-class* constructor or specified *class-desc*. If neither supplied, then item type inferred using initial items, if no items then desired type used and if desired type not known then **Object** used.

7 [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. Second optional '^' indicates scope of surrounding context used (i.e. refers to surrounding invoked object directly – which may go out of scope before this closure) rather than making a reference copy of any captured variables. 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.

8 **[first]..[[.]last]|(#count)** Range from initial inclusive expression value (0/default? if omitted) to second exclusive expression value (-1/Type.max? if omitted, inclusive if optional third '.' used). If '#' used then until first expression + second expression. If neither expression is specified and the desired type is not known then **Integer** type is inferred.

9 Syntax sugar/optimization of *closure* – gets all information such as interface from receiver object and single method/coroutine.

10 Key-value types determined via optional *map-class* constructor or specified key-value *class-desc* types. If neither supplied, then key-value types inferred using initial *key-value* pairs, if no pairs then desired type used and if desired type not known then **Object** used for both key and value types.

11 If desired enumeration class type can be inferred (like when passed as an argument) then optional enum-class may be omitted.

Identifiers:

identifier[1]	=	*variable-identifier* | *reserved-identifier* | *class-identifier* | *object-id* | *routine-identifier*
variable-identifier[2]	=	*variable-name* | ([*expression* *ws* '`.`' *ws*] *data-name*)
variable-name	=	*name-predicate*
data-name[3]	=	'`@`' | '`@@`' *variable-name*
reserved-identifier	=	'`nil`' | '`this`' | '`this_class`' | '`this_code`'
class-identifier	=	*class-name* | *enum-class* | *flagset-class*
object-id[4]	=	[*class-name*] '`@`' ['`?`' | '`=`'] *symbol-literal*
invoke-name	=	*method-name* | *coroutine-name*
method-name[5]	=	*name-predicate* | *constructor-name* | *destructor-name* | *class-name* | *binary-operator* | *postfix-operator*
name-predicate[6]	=	*instance-name* ['`?`']
constructor-name	=	'`!`' [*instance-name*]
destructor-name[7]	=	'`!!`'
coroutine-name	=	'`_`' *instance-name*
instance-name	=	*lowercase* {*alphanumeric*}
class-name	=	*uppercase* {*alphanumeric*}
routine-identifier	=	'`@`' ([*expression*] '`.`') | *scope* *invoke-name*

Flow Control:

flow-control	=	*code-block* | *loop* | *loop-exit* | *loop-skip* | *conditional* | *case* | *when* | *unless* | *concurrent-block* | *class-cast* | *class-conversion* | *query-cast* | *proviso*
code-block	=	'`[`' *ws* [*expression* {*wsr expression*} *ws*] '`]`'
loop[8]	=	'`loop`' [*ws* *instance-name*] *ws* *code-block*
loop-exit[9]	=	'`exit`' [*ws* *instance-name*]
loop-skip[10]	=	'`skip`' [*ws* *instance-name*]
conditional	=	'`if`' {*ws* *expression* *ws* *code-block*}$^{1+}$ [*ws* *else-block*]
case	=	'`case`' *ws* *expression* {*ws* *test-expr* *ws* *code-block*}$^{1+}$ [*ws* *else-block*]
else-block	=	'`else`' *ws* *code-block*
test-expr	=	*case-operand* {*ws* ['`,`' *ws*] *case-operand*}$^{1+}$
case-operand	=	*expression* | *range-literal*
when	=	*expression* *ws* '`when`' *ws* *expression*
unless	=	*expression* *ws* '`unless`' *ws* *expression*
concurrent-block	=	*sync* | *race* | *rush* | *fork* | *branch* | *divert*
sync[11]	=	'`sync`' *ws* *code-block*
race[12]	=	'`race`' *ws* *code-block*
branch[13]	=	'`branch`' *ws* *code-block*
divert[1]	=	'`divert`' *ws* *code-block*

1 Scoping not necessary – instance names may not be overridden and classes and implicit identifiers effectively have global scope.

2 Optional *expression* can be used to access data member from an object – if omitted, `this` is inferred.

3 '`@`' indicates instance data member and '`@@`' indicates class instance data member.

4 If *class-name* absent **`Actor`** inferred or desired type if known. Optional '`?`' indicates result may be **`nil`** – if question mark not used and object not found at runtime then assertion error occurs. Optional '`=`' indicates a symbol literal validated by class type.

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

6 Optional '`?`' used as convention to indicate predicate variable or method of return type **`Boolean`** (**`true`** or **`false`**).

7 Destructor calls are only valid in the scope of another destructor's code block. *[Ensure compiler check.]*

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

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

10 Restarts/continues loop by jumping to loop start – 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 1+ durational expressions run concurrently and diverted (updating on receivers rather than current updater – see *divert*) and the next *expression* executed immediately (result **`nil`**). Essentially: **`fork [divert [expr]]`**

rush[2]	=	‘**rush**’ *ws code-block*
fork[3]	=	‘**fork**’ *ws code-block*
query-cast[4]	=	*expression ws* ‘**<?>**’ {*ws class-desc* [*ws code-block*]}$^{1+}$ [*ws else-block*]
proviso[5]	=	‘**\\proviso**’ *wsr proviso-test ws code-block*
proviso-test[6]	=	*instance-name* | (‘**[**’ *proviso-test* ‘**]**’) | *operator-call*

Primitives:

primitive	=	*create-temporary* | *bind* | *class-cast* | *class-conversion* | *nil-coalesce* | *list-expansion*
create-temporary	=	‘**!**’ *ws variable-name* [*ws binding*]
bind[7]	=	*variable-identifier ws binding*
binding[8]	=	‘**:**’ *ws expression*
class-cast[9]	=	*expression ws* ‘**<>**’ [*class-desc*]
class-conversion[10]	=	*expression ws* ‘**>>**’ [*class-name*]
nil-coalesce[11]	=	*expression ws* ‘**??**’ *ws expression*
list-expansion	=	‘**%**’ *expression*

Invocations:

invocation	=	*invoke-call* | *invoke-cascade* | *apply-operator* | *invoke-operator* | *index-operator* | *slice-operator* | *instantiation*
invoke-call[12]	=	([*expression ws* ‘**.**’ *ws*] *invoke-selector*) | *operator-call*
invoke-cascade	=	*expression ws* ‘**.**’ *ws* ‘**[**’ {*ws invoke-selector* | *operator-selector*}$^{2+}$ *ws* ‘**]**’
apply-operator[13]	=	*expression ws* ‘**%**’ | ‘**%>**’ | ‘**%,**’ | ‘**%<**’ | ‘**%.**’ *invoke-selector*
invoke-operator[14]	=	*expression bracketed-args*
index-operator[15]	=	*expression* ‘**{**’ *ws expression ws* ‘**}**’ [*ws binding*]
slice-operator[16]	=	*expression* ‘**{**’ *ws range-literal* [*wsr expression*] *ws* ‘**}**’
instantiation[17]	=	*class-instance* | *expression* ‘**!**’ [*instance-name*] *invocation-args*

---

1 Durational expressions to be maintained on receiver updater object rather than by the calling updater object.

2 Like *race* except: return args bound in expression completion order and other expressions continue until *completed*.

3 *code-block* 1+ durational expressions run concurrently and next *expression* executed immediately (result **nil** with return args bound in order of expression completion???) – turns block of durational expressions into immediate.

4 if *expression* is a *variable-identifier* its type is modified in any matching clause block. If a clause block is omitted the result of *expression* is cast to the matching type and given as a result.

5 Conditional code that will be compiled only if *proviso-test* evaluates to true. [Alternatively, this could be structured like a *conditional expression* with 1+ test clauses and an optional “else” clause.]

6 *instance-name* refers to set of predefined proviso labels – example “debug”, “extra_check”, etc. [It could be any valid Boolean *expression* – with limits based on availability of code at compile time.] *operator-call* uses *proviso-test* rather than *expression*.

7 [Consider: Make *bind* valid only in a *code-block* so that it is not confused in *key-value* for *map-literal*.]

8 [Stylisticly prefer no *ws* prior to ‘**:**’ – though not enforcing it via compiler.]

9 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. Release: no code generated.]

10 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”**.

11 **expr1??expr2** is essentially equivalent to **if expr1.nil? [expr2] else [expr1<>TypeNoneRemoved]**.

12 If an *invoke-call*'s optional *expression* (the receiver) is omitted, ‘**this.**’ is implicitly inferred. *[Consider whitespace.]*

13 If **List**, each item (or none if empty) sent call – coroutines called using **%** – **sync**, **%>** – **race**, **%,** – **rush**, **%<** – **fork**, **%.** – **span** 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.

14 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.

15 Gets item (or sets item if *binding* present) at specified index [may be negative – see *slice-operator*].

16 Returns **Integer** sub-range: **{[first]..[[.]last]|(#count)[step]}**. Where: **last** and **first** may be negative with -1 last item, -2 penultimate item, etc.; **step** may be negative indicating sub-range in reverse order.

17 *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**.

```
invoke-selector       =  [scope] invoke-name invocation-args
scope                 =  class-unary '@'
operator-call[1]      =  (prefix-operator ws expression) | (expression ws operator-selector)
operator-selector     =  postfix-operator | (binary-operator ws expression)
prefix-operator[2]    =  'not' | '-'
binary-operator       =  math-operator | compare-op | logical-operator | ':='
math-operator[3]      =  '+' | '+=' | '-' | '-=' | '*' | '*=' | '/' | '/='
compare-op            =  '=' | '~=' | '>' | '>=' | '<' | '<='
logical-operator[4]   =  'and' | 'or' | 'xor' | 'nand' | 'nor' | 'nxor'
postfix-operator      =  '++' | '--'
invocation-args[5]    =  [bracketed-args] | closure-tail-args
bracketed-args        =  '(' ws [send-args ws] [';' ws return-args ws] ')'
closure-tail-args[6]  =  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            =  [named-spec ws] variable-identifier | create-temporary
named-spec[7]         =  variable-name '#'
```

Parameters:

```
parameters[8]         =  parameter-list [ws class-desc] ['!']
parameter-list        =  '(' ws [send-params ws] [';' ws return-params ws] ')'
send-params           =  parameter {ws [',' ws] parameter}
return-params         =  return-param {ws [',' ws] return-param}
parameter             =  unary-param | group-param
return-param          =  param-specifier | group-specifier
unary-param[9]        =  param-specifier [ws binding]
param-specifier[10]   =  [class-desc ['!'] wsr] variable-name
group-param[11]       =  group-specifier [ws binding]
group-specifier[12]   =  '{' ws [class-desc {wsr class-desc} ws] '}' [digits] ws instance-name
```

1 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 **]**).

2 See math-operator footnote about subtract on how to differentiate from a negation '-' prefix operator.

3 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.

4 Like other identifiers – whitespace is required when next to other identifier characters.

5 *bracketed-args* may be omitted if the invocation can have zero arguments

6 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.

7 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.

8 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. '!' indicates result returned by value (**!copy()** is called on it) rather than just being returned by reference.

9 The optional *binding* indicates the parameter has a default argument (i.e. supplied *expression*) when argument is omitted. ':' uses instance scope and '::' indicates calling scope used to evaluate the default.

10 '!' indicates arguments passed by value (**!copy()** is called on them) rather than just being passed by reference. 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**.

11 If default binding is omitted an empty list is used as the default.

12 **Object** inferred if no classes specified. Class of resulting list bound to *instance-name* is class union of all classes specified. The optional *digits* indicates the minimum number of arguments that must be present.

Class Descriptors:

class-desc	=	class-unary | class-union | nested-enum | label
class-unary	=	class-instance | meta-class | enum-class | flagset-class
class-instance	=	class | list-class | invoke-class | map-class | code-class
meta-class	=	'<' class-name '>'
class-union[1]	=	'<' class-unary {'|' class-unary}1+ '>'
invoke-class[2]	=	['_' | '+'] parameters
list-class[3]	=	`List` '{' ws [class-desc ws] '}'
map-class[4]	=	`Map` '{' ws [class-desc] ':' ws [class-desc ws] '}'
code-class[5]	=	[class-unary ws] '.' invoke-class
enum-class[6]	=	[class-name ['@' invoke-name]] enumeration-name
nested-enum	=	'#' | enumeration-name ws enumerator-list
label	=	'#' '`Symbol`' | '`String`'
flagset-class	=	[class-name] flagset-name
flagset-name	=	'##' alphabetic {alphanumeric}

Whitespace:

wsr[7]	=	{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 | parser-comment
single-comment	=	'//' {`printable`} end-of-line
multi-comment	=	'/*' {`printable`} [multi-comment {`printable`}] '*/'
parser-comment[8]	=	'\\' *parser-hint* end-of-line

Characters and Digits:

character	=	escape-sequence | `printable`
escape-sequence[9]	=	'\' 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 Indicates that the class is any one of the classes specified and which in particular is not known at compile time.

2 '_' 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.

3 `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.

4 `Map` is any **`Map`** derived class. If *class-desc* in key/value class descriptors is omitted, **`Object`** inferred when used as type or types are deduced when used with *map-literal*. A *map-class* of any key/value type can be passed to simple untyped **`Map`** class.

5 Optional *class-unary* is the receiver type of the method/coroutine – if it is omitted then **`Object`** is inferred.

6 Optional *class-name* and *invoke-name* qualification only needed if it cannot be inferred from the context – so it may be omitted and inferred if inside the required scope or if the expected enumeration class type is known, etc.

7 *wsr* is an abbreviation for (w)hite (s)pace (r)equired.

8 [Consider different compiler hints – ex: disable warning X. Should also be a way to hook in application custom compiler hints.]

9 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 '''. Also see *escaped-string*.