Appendix A: zkl Grammar

Ah yes, the dreaded Grammar. Watch the eyes start to glaze over, like the goo on a hot-out-of-the-oven Krispy Krème.
-- Zander Kale

The following grammar is quite loose, it is not formal nor complete; it is intended to be a "how to" guide to combining elements into a program. Many things are undefined or incompletely defined here, look for complete definitions in the relevant sections of the manual.

Fonts: `keyword`, **`code literals`**, *defined somewhere in here*, ***concept definition***

Concepts

program := *block* (the enclosing brackets are usually implicit)

object := The turtle that the rest of the world resides on. A virtual concept, as only instances of objects exist.

instance := An instance of a o*bject*. Examples: *number*, *string*, *class*.
instance := *class* | *fcn* | *method* | …
instance := *instance*([p*arameters*]); Instance creation: call the objects "create" method or creates a new copy of a class and call the constructor and init functions.
instance := object *name* (such as List, L, Atomic, etc)

attributes := **[** attribute-name [,name …] **]** Attributes are placed between a type and name and apply to all the following names.

- For example: `var [const] v;`
- A comma between attributes is optional; `var [private,const]` is the same as `var [private const]`.

block := **{** *block* | *expression* | *keyword* | *object* | *assignment* | *listAssigment* | *mathSet* **}**

expression := *stuff* | **(** *stuff* **)** | **(** *stuff* **).***dataRef* [more *stuff*]

stuff := *instance* | `if` | *dataRef* | `try` | *assignment* | *mathSet* | *call* | *math* | *logic* | `throw` | *listAssigment* | *number* | *string* | `switch` | `fcn` | `:` | `(` | `)`

- A something can be forced/cast to an expression by enclosing something in ()s. For example, `if(x) y;` isn't an expression[1] but `(if(x) y;)` is.
- `(`*`stuff1`*`)(`*`stuff2`*`)` *should* evaluate to a call to stuff1 with parameters stuff2 but doesn't.
- `(`*`stuff1`*`)`*space*`(`*`stuff2`*`)` is the same as `(`*`stuff1`*`);(`*`stuff2`*`)`.
- `( (`*`stuff1`*`)`*space*`(`*`stuff2`*`) )` doesn't work.

control := ***(*** `if` | *dataRef* | `try` | *assignment* | *mathSet* | *call* | *math* | *logic* | *control* | `switch` | `(` | `)` | `:` ***)***

- A mini **(***expression***)**
- `if( (sideEffects) (`*`control`*`) )` *should* work but doesn't.

call := `fcn` | `class` | *method* | *instance* | *constName***(**[*parameter [, parameter …]]***)** *terminator*

- No space between object and `(`
- Any object that has a create method (and most do) can be called, to create a new instance or run the instance.

parameter := `if` | *dataRef* | *assignment* | *mathSet* | *call* | *math* | *logic* | `switch` | `try` | `(` | `)` | `:` | *`fcn`* | *classDefinition*

- A mini *expression* plus functions and classes
- `f( (`*`sideEffects`*`) (`*`arg`*`) )` *should* work but doesn't.

callable := Any instance that is referenced by a *call*.

name := [0-9a-zA-Z_]+ No more than 80 characters.

- It is a bad idea to use a single "_" as a name, as it is often used as a "replace me" symbol.
- The compiler will use two leading underscores and a trailing "#" for "private" symbols (eg "__fcn#1").

Class := The Class o*bject*, a virtual concept. A container for variables and functions.
class := An instance of a *Class*
class := *class*([*parameters*]). Class creation: create a copy and call the class constructor and init functions (in that order). This can be overridden in various ways.

- `class {…}(x)` both defines and creates a new instance of the class.
- `class {…}.M` defines a class and resolves M against that class.

RootClass := The class that encloses all classes (except itself), eg a source code file.
RunMeClass := A *class* with a runMe function, which overrides class creation.

comment := `#` rest of line ignored
comment := `//` rest of line ignored

1 Yes, this contradicts *stuff*. What happens is that single a object doesn't (usually) need the expression wrapper so the wrapper is thrown away when parsing.

comment := **`/*`** comment **`*/`**, can be nested: /* /* */ */
comment := **`#if`** 0|1|*name* \n comment \n **`#endif`**, can be nested, see below
comment := **`#ifdef`** *name* → `#if 1` if name has been defined, else `#if 0`
comment := **`#define`** *name* 0|1 For use with #ifdef
comment := **`#fcn`** *name* { *body* } For use with #tokenize.
comment := **`#text`** *name text* For use with #tokenize
comment := **`#tokenize`** *name|f|f(parameters)* Evaluate name or function and tokenize the result as a string (as if it were part of the source at that point).
#cmd comments can't be proceeded by anything but space, otherwise they are treated as a comment.

Delimiter := *`; , : ( ) { } = ! + - * / % < > " #` whitespace newline*

dataRef := Data Reference. A data description. Data is any object.
dataRef := root.next.next...
dataRef := *dataRef2* | *dataRef3*
dataRef2 := variable*Name* | constant*Name terminator*
dateRef2 := *instance.instance* | *instance.dataRef2 terminator*
dataRef2 := *instance.property terminator*
dataRef2 := *dataRef2([parameters]) terminator*
dataRef2 := *dataRef.dataRef2 terminator*
dataRef3 := ***(****stuff****)****.dataRef2* | ***(****stuff****)(****parameters****)[****.dataRef2] terminator*

- Whitespace (including newline) is OK in front of a dot. Not OK after a dot. For example: `foo.bar` is the same as `foo .bar`

assignment := variable = *fcnDefinition* | *classDefinition* | *expression* | *block terminator*
multiple assignment := variable = variable … *assignment*
list assignment := variable, variable [, …] = *expression terminator*
list assignment := *variable, variable [, …] := expression terminator*

- Expression must evaluate to an object that supports [] (List being the most popular).
- The destinations must exist (as registers, variables or some combination), unless it is "_".
- If := is used, registers are created (if they don't exist in the enclosing block).
- If "_" is one of the destinations, it is thrown away.

= := reg/var = *expression*. See assignment. The destination must be an existing variable or register (although it may be defined after the assignment).
:= := reg **:=** *expression*

- := is assignment and is the same as = except that it is restricted to registers. If *reg* doesn't exist, it is created in this block. `x:=5;` acts like `reg x:=5;` If *reg* does exist, := is the same as = (`reg r; r:=5;` is the same as `reg r=5;` or `reg r; r=5;`).
- `r1:=r2:=e;` is the same as `r1:=e; r2:=e;` (e evaluated only once).
- `R1:= v= r2:=e` is the same as `r1:=e; v=e; r2:=e;`
- List assignment: `r1,r2,_:=e;` creates/reuses registers r1, r2 and ignores e[2].

fcn := A function object, a code container. Often contained in a class. A static function can be homeless. All functions are first class objects.
lambda := `fcn { … }`, an anonymous function

- `fcn {…}(x)` both defines and calls the function.
- `fcn {…}.M` defines a function and resolves M against that function.

method := The object equivalent of a function.
property := Passive read-only data attached to a object, roughly equivalent to a read-only class variable.

number := integer | 0x[0-9a-zA-Z]+ | float (123.0 | 1.23 | 0.123 | 1e*N* | 1E*N*)
string := "text" | "one" "two" … Adjacent string constants are concatenated.
"\\ \b \f \n \r \t" convert to backslash, backspace, formfeed, newline, return, tab
"\xHH" converts two hex digits into one ASCII character.
"\uHHHH" converts a four [hex] digit Unicode character to a two to three byte UTF-8 character.
string := 0' *sentinel text sentinel* Raw string. Examples: 0'|text|, 0'"text"

terminator := `;` | `{` | `}`

logic := `and` | `or` | `not`

mathSet := object *op*= *expression terminator*

- `a += 1` → `a = a+1`
- The object has to be a singleton, `a.b += 1` is illegal.

math := [-] object [*op math*]

op := `+` | `-` | `*` | `/` | `%`

[] := *dataref*`[…]` → *dataref*`.__sGet(…)`
dataref`[…]=x` → *dataref*`.__sSet(x,…)`

Keywords

keyword : =
`AKA` | `Attributes` | `break` | `catch` | `class` | `const` | `continue` | `critical` | `_debug_` | `do` | `fcn` | `foreach` | `if` | `include` | `onExit` | `onExitBlock` | `reg` | `return` | `returnClass` | `self` | `switch` | `throw` | `try` | `var` | `while`

class := `class` [*attributes*] [*name*][**(***parent(s)***)**] *block*
attributes := noChildren | private | public | static

- A static class never has more than one instance (the reference or Eve instance).
- Nobody can inherit from a noChildren class.

- A private class is not visible outside of the file it is defined in (ie is only source code can see it). It can be seen with reflection.

: (***compose***) := *expression* with one or more colons in it.
Compose works like so: Given an expression E = E_1:E_2:...:E_n and pseudo variable X, the result of the composition is X = E_1; X = E_2(X); … X = E_n(X);
The position of X in E_n is marked by a underscore. $E_{n>1}$ has the following constraints:

- It must have a call (eg function or method). Unless:
- The only time _ isn't a parameter is when assigned (`:(x=_)`) or `:(_).`*`name`*.
- Can't be too complex (whatever that means).

See Keywords.: (compose)

const := Parse time constant, must evaluate to a "simple" constant
const := `const` *name* `=` *expression* | *fcnDefinition* | *block*
const := `const` *nameSpace* `{` *const* `}`

- Blocks and expressions are evaluated at parse time (after tokenizing and before compilation), as are calls to const functions

`False` := A boolean value. There is only one.

fcn := `fcn` [*attributes*] [*name*][**`(`***prototype***`)`**] *block*

- Define an instance of a Function object.

prototype := *name* | *name=parameter* …
attributes := private | public

- A private function will not be visible when in a class (but can be found with reflection).
- A static function is one that doesn't reference any instance data. The compiler determines this attribute.

if := `if(`*control*`)` *expression terminator*
if := `if(`*control*`)` *block*
if := `if` … `else` *expression* | *block [terminator]*
if := `if` … `else if(`*control*`)` *expression* | *block* [`else if` …]
if := `if` … `else if` … `else` …

- If an if is part of an expression, such as `5 + if(a<b) b-a else a-b`, you will often have to terminate both the if and the expression, depending where the if falls in the expression.

```
5 + if(a<b) b-a else a-b;;  a+b
5 + if(a<b) b-a else {a-b}; a+b
```

switch := `switch(`*control*`)` *block*
switch := `switch(`*control*`) {case(…)` *block* … `else` *defaultBlock* `}`
switch := `switch(`*control*`) { case(…) [fallthrough]` *block* … `}`

loop := `while (`*control*`)` *block* [`fallthrough` *block*]
loop := `do` *block* `while(`*control*`)`

loop := `do(`*n*`)` *block*
- *block* can contain `break` and/or `continue`

loop := `foreach` *n* `in` *(object)* *block* [`fallthrough` *block*]
loop := `foreach` *a,b,c* `in` *(x)* *block* [`fallthrough` *block*]
loop := `foreach` *a,b,c* `in` *(x,y,z)* *block*
- *n* is is the name of the control register and is created local to block.
- *Object* needs to have a function or method (named `walk`) that returns a Walker, which most objects do (lists, files, strings, numbers, etc). Since all Walkers have a walk function (which returns the Walker), you can explicitly create a Walker.
- "__*n*Walker" is a register created for access to the walker.
- List assignment works: `foreach a,b,c in (…)`
- A cascading foreach (`foreach a,b,c in (x,y,z)`) is the same as
```
foreach a in (x){ foreach b in (y){ foreach c in (z)
   { block }}}
```

return := Exit fcn with value. Optional (fcn result is then the result of the last calculation).
`return()` | `return(value)` | `return(value, value, …)`
`return()` → `return(Void)`
`return(value, value, …)` → `return(List(value, value, …))`
()'s are required and no space between return and (
returnClass := `returnClass(`*object*`)`
- The same as return except that returnClass can be used in a constructor or init function. One, and only one, parameter is required. Of course, that object can be a list.

try := `try` *block* `catch` *block* [`else` *block*]
try := `try` *block* `catch` *block* `catch` *block* … [`fallthrough` *block*]
catch := `catch` | `catch(`*exceptionName [,name ...]*`)`
catch := `catch(+trace, -trace)` | `catch(`*name*`,+trace)` | `catch(+trace,`*name*`)`

critical := `critical` *block*
critical := `critical(`*lockName*`)` *block*
critical := `critical(`*lockName*, *acquireName*, *releaseName*`)` *block*
- Restrict execution of block to a single thread. A lock is allocated if need be, otherwise, *lockName* is the name of a var that holds a locking object (such as Atomic.Lock or Atomic.WriteLock).

critical := `critical(`*object*, *name1*, *name2*`)` *block*
- Expand to *`object.name1(); block object.name2();`*
  No locking or thread safeing is done.

onExit := `onExit(f [,parameters])` ≃ `Deferred.once(f,parameters)`
- When the enclosing function returns, run exit code, equivalent to a "finally" block for a function. The code is always run.

onExitBlock := `onExitBlock(f [,parameters])`
- When the enclosing block exits, run exit code, equivalent to a "finally" block for a block. The code is always run.

Tailcalls can change *when* you think the exit code will run.

reg := A var in the current scope (block) and are not part of the instance. See :=.

`True` := A boolean value. There is only one.

var := `var` [*attributes*] *name*
var := `var` *name* [, *name* ...]
var := `var` *name* = *value* …
var := `var` *name* = (*expression*) ...
var := `var` *name* = `fcn` *block* ...
var := `var` *name* = `class` *block* ...
attributes := `const` | `mixin[= class or Vault object]` | `private` | `protected` | `proxy`, separated by commas or spaces.

- Constant variables can only be set during declaration.
  For example: `var [const] v=5;`
- `var [mixin] m=List;` is the same as `var [mixin=L] m;` This allows mixin checking to used where the type can't be determined at compile time:
  `var [mixin=Op] op=Op("+");`

`Void` := An object that doesn't do much. Like a nil or null, only more so. There is only one.

Comments

There are three types of comment: to end of line, block and a combination of the two. Comment characters in strings are ignored and the characters that start a comment must be preceded by white space (or the start of a line or other delimiter). Comments are recognized by the tokenizer and never reach the parse stage.

- C++ "//" ignore to end of line type: `//` text
- Shell "#" ignore to end of line type: # text
- C type "/* */" block type: /* text */
  Text can span multiple lines. Single line comments are active and can preempt the closing comment `/* // */` and `/* # */` are errors but `/* // */ two line comment */` is OK. Strings are also active so `/* "*/" */` is valid as is `/* "//" */`. The "*/" does not need to be preceded by white space (`/**/`, `/**********/` and `/*foo*/` are valid).
  Block comments can be nested.
  The text in a block comment is tokenized so it must be syntactically valid (or just enough so that the tokenizer can recognize the end of the comment).
- C preprocessor type: #if *value*|*name* #else #endif
  This type overloads shell comments to become block comments. This style is only valid if the cmd word is at the start of a line (optionally preceded by white space) and all the characters are in a single word (ie `#if` and not `# if`). Otherwise, this is just another line

comment and, as such, interacts the in same way with the above comments (specifically, /* can hide #endif).
Value must be zero (0), one (1) or something previously created by #define.
It is an error if *name* hasn't been defined.

- #ifdef *name* #else #endif
  Name is something that may have been created by #define. If it was, the #ifdef becomes "#if 1", otherwise, "#if 0".
- #define *name* 0|1: Associate a name with zero or one for use with #if. #define doesn't span compilation units (eg files) and thus isn't seen in included files and vice versa.
  Name must be a valid name (valid is defined elsewhere).
- See the concepts section for #fcn, #text and #tokenize.

Data Reference Resolution

The compiler resolves references to data (objects) using a "look up, look down" algorithm:

A. Search upwards for the "root". The root is the object in the class hierarchy that matches the first element of the data reference.

1. First, is the first element:
   - a const, number, string, expression
   - `self`: The root is the enclosing class definition
   - `self.fcn`: The root is the enclosing function definition
   - `TheVault`: The search starts in the Vault
   - If any of the above, goto B
2. The block is examined for:
   - A parameter (if started in a fcn and still in that fcn).
     In `fcn f(a){a.len()}`, the root of `a.len()` is parameter a.
   - A function, class, register, variable, parent, call:
     - If a call, the result of the call replaces the stuff up to that point.
       `"foo".len().type` becomes `3.type` → "Int"
     - The value replaces name:
       `class C{var v="foo";} C.v.len()` → 3
       Name "v" is replaced with value "foo": `"foo".len()`
   - If match, goto B
3. If the top of this class definition is reached (the class that encloses the data reference):
   - The parents are searched in a breadth first search for [class] instance data (parent, class, function or variable).
   - The methods and properties are searched[2]
     `dir()` → `self.dir()`
     A match here might not be the end of the line: `name.len()`
   - If match, goto B
   - the search moves to the enclosing block
   - goto 2

2 But print, println and ask are ignored as they can be both Object methods and sugar. Sugar wins here.

4. If the root class (ie the source code file) is reached:
    - The Vault is checked (eg File, Test.UnitTester)
    - Syntactic sugar is checked (println, ask, etc)
    - If not found, it is a syntax error
    - goto B
5. The search moves to the enclosing block, goto 2

B. Resolve down:
The next item is resolved relative to the previous item.
Call, class, function, variable, parent, method, property
References are statically bound to the point where it becomes ambiguous[3] and late bound from that point on.

Attributes: const, private, protected, proxy

- If a variable has attribute **const**, it acts like a read-only variable with the restriction that it can only be set once, and that once has to be in the var statement. If not set, it will forever be Void. There is an exception to this rule: const variables in functions (such as init), are always set when the function is run.
- **Private** variables, functions and classes are visible in the compilation unit (usually file) they are defined in. Class.resolve won't find them, thus they are invisible outside of the compilation unit although they can be seen with reflection (eg class.fcns). This is analogous to static functions in C. In C++, this is between protected and private.
    - Anonymous functions and classes are private.
    - Private variables are anonymous; they share the attributes of private functions but can't be found with reflection.
    - If a class inherits from a compiled class, the new class won't be able to reference the private objects in the parent.
- A **protected** variable can only be set if it is the first word in a data reference or if the reference is to a protected parent variable.
```
var [protected] v; v = 3;                 // OK
class C { var [protected] v; } C.v = 3;   // error
class D(C) { C.v = 3; }                   // OK
```
  This effectively restricts writing to the class they are defined in, containing classes and parents. These variables are visible to everybody (unless marked private).
- A **proxy** variable is an active or trampoline variable; referencing it causes an action to happen. For example
```
var [proxy] pv = fcn {"test"};  pv; → "test"
```
  The action is always `pv()`, where pv is the proxy variable.

Expressions

An expression is a group of operations that return a result. Expressions can do many things besides math; data references, function calls, try/catch, etc. Sometimes an expression needs to be wrapped in parentheses "()" to avoid ambiguity. Here is the algorithm:

1. Keep track of opening "(" and closing ")" parentheses

3 The definition of ambiguous is somewhat ambiguous.

2. Check for (*).*. For example `("1"+"2").len()` → `("12").len()` → 2
3. Check for "if". Example: `x:=(if (a==3) 56 else 65);`
4. Check for "try". Example: `x=try{a/0}catch{"division by zero throws"}`[4]
5. Check for "break" and "continue": `while(1){if (x==3) break}`
6. Check for the different types of assignment: `a=2; a+=2; a,b,c=f();`
7. Check for a data reference
8. Check for a call (function, method, creation, etc)
9. Evaluate the "normal" math stuff: +,-,*,/,%, ==,!=, <,<=,>,>=, not, and, or, unary minus. There is nothing special here, and objects can overload. The precedence rules are almost the same as C, with the exception that evaluation is always left to right.

Operators (precedence: high to low)	C Equivalent
() (function call) .(dot, resolve) [] ( ) (grouping)	() . -> [] ( )
not, unary minus	! -
* / % (modulo)	* / %
+ -	+ -
< <= > >=	< <= > >=
== !=	== !=
and or (differs from C)	&& ||

"or" and "and" evaluate their result as a boolean but don't produce a boolean. For example, `(1 and 2)` evaluates to 2 but behaves as if it were True (since `(2).toBool()` returns True). They also "short circuit", which means they only evaluate as much of an expression as they need to. For example, `(a and 0 and b)` can be reduced to `(a and 0)` (which is `(0)` unless a blows) and `(a or 0 or b)` can be reduced to `(a)`. You can take advantage of this in calculations.

- To access a list only if it has elements: `(list and list[0])`. This returns the empty list if list is L() (since `L().toBool()` is False) and the first item if list isn't empty[5].
- Code crunch: `f() or throw(Exception.BadDay)` throws an exception if f() doesn't return a "positive" value. This is the same as
  `if (not f()) throw(Exception.BadDay)`

And factorial can be written: `fcn(x){ x and x*self.fcn(x-1) or 1 }`

Assignment (=, +=, -=, *=, /=) precedence is lower than grouping but is otherwise somewhat ambiguous. If you use "=", etc in a mixed expression, you almost always want to wrap it in ():
`if ((n=f()) != 4) doSomething(n)`
Without the parentheses, this would evaluate as `if(n=(f()!=4))`, which is a Bool, not a number (assuming f always returns a number).

4 An interesting thing about this example is what happens if the try succeeds. For example, if a == 2 and the try is { a/1 }, then x is set to 2.

5 An even easier way to do this: `list[0,1]`

In `n=3+4`, n→7; `(n=3)+4`, n→3; `3+n=4`, n→4; the sum is always 7. *When* n is set is not specified.

Compose (:) "chunks" an expression; each chunk is evaluated and rolled into the next chunk, forming a new chunk.

Ambiguities, or, what are you trying to express?
If the parser see a "(", it tries to evaluate an expression. Otherwise, it looks for a statement. Failing that, an expression (which, in turn, will look for some statements).
Now, what does `if(X) 1 else 2; +5` mean? "If" is a statement, + 5 is an expression. But you can't add a statement and expression so it means an error message. By adding ()s, it can be forced/cast to an expression: `(if(X) 1 else 2; +5)` or `(if(X) 1 else 2) +5`
In the first case, a ";" is needed to terminate the if statement.
This doesn't need ()s (it doesn't start with "(", it isn't a statement, maybe an expression): `5 + if(X) 1 else 2;; println("ick")` but it does need two semis, one to terminate the if and the second to terminate the expression. So you may prefer ()s and a single semi. Unless you have something like: `5 + if(X) 1 else 2; +6; println("12 or 13")` which really looks better with ()s: `5 + (if(X) 1 else 2) +6;` If you are writing code you'll have to read later, it will probably make your life easier to use ()s if in doubt.

Chained Compares
Comparison operators (==, !=, <, <=, >, >=) can be chained. For example,
`A==B==C` → `A==B and B==C`, where B is evaluated only once. Here are the rules:

- Only three terms (A,B,C) per chain.
- Terms can be anything that is legal in the expression.
  Eg `A == try { B } catch { C } != f(5); A<=(f(5)+1)>B`
- Precedence can't be mixed. Eg `A==B<C` is not legal.
- Evaluation is left to right.
- and/or terminate a chain: `A==B==C and E==F==G` is two chains.
- ()s scope. Not very useful. `(A==B)==C` → `Bool==C`

Scoping

zkl is block (lexically) scoped; if something is created in a block, it stays there, unless explicitly moved to a enclosing block. Classes, functions and variables are scoped to the class or function they are created in. On occasion, something created outside a block will be moved into a block. For example, the loop variable in a foreach statement is moved into the loop block. And, vice versa, a variable is will migrate to nearest enclosing class scope, while a register sticks to its block.

> *It's all very complicated and would take a scientist to explain it.*
> -- Mystery Science Theater 3000