CHAPTER 1

The E Kernel Language Reference Manual

The E language is specified in layers. At the bottom is the E kernel language. The kernel language is a subset of the regular E language -- every program written in Kernel E is also a valid E program with the same meaning. The remainder of E's grammar outside the kernel subset is E's sugar (see *The E Language Grammer*). The semantics of the sugar is defined by canonical expansion to Kernel E. This expansion happens during parsing -- E parse trees only contain nodes defined by Kernel E -- so only these are executed by the virtual machine.

To give a semantics of Kernel E it suffices to write an executable specification of the virtual machine as an interpreter of such parse trees. Following a venerable tail-biting tradition, this chapter presents such an interpreter written in the full E language. (An interpreter written in the same language it interprets is called a *Meta-Interpreter.*)

Unfortunately, this does cause some circular-definition ambiguity. In Brian Smith's terminology[?], the interpreter *absorbs* some issues by mapping them onto the same issues in the language in which the interpreter is written. When these are the same languages, this leaves some issues unresolved. For the moment, we resolve these ambiguities only informally in the text. The bootstrap E interpreter is essentially a transliteration into Java of the interpreter presented here.

Since this chapter is not concerned about surface syntax, the BNF statements of the kernel productions leave out non-structural grammatical detail, such as issues of precedence and associativity. See *The E Language Grammar* for these.

E Kernel Language Quick Reference Card

In the pseudo-BNF used here, *Terminals* (tokens emitted by the lexer) are either quoted bold-faced strings, or are names that begin with an upper-case letter. *Non-terminals* (forms defined by this grammar) are names that begin with a lower-case letter. A question mark suffix on a form means the appearance of the form is optional. An asterisk suffix means zero or more repetitions of that form. If the asterisk is immediately followed by a punctuation character, these repetitions are *separated* by that puntuation character.

Helper Productions

```
verb:           Identifier

method:         “to” verb “(” pattern*, “)” “{“
                    expr
                “}”
matcher:        “match” pattern “{“
                    expr
                “}”
```

pattern: *any of the following*

```
param:          definer | ignore
definer:        Identifier “:” expr
ignore:         “_”

suchThat:       pattern “?” expr
tuple:          “[” pattern*, “]” “+” pattern
```

expr: *any of the following*

```
literal:          BigIntegerLiteral | DoubleLiteral
                | StringLiteral | CharLiteral

noun:            Identifier
slot:            “&” noun
assign:          noun “:=” expr
sequence:        expr “;” expr
call:            expr verb “(” expr*, “)”
send:            expr “<-” verb “(” expr*, “)”
matchBind:       expr “=~” pattern
init:            “define” pattern “:=” expr
methodical:      “define” param “{”
                     method*
                     matcher?
                 “}”
plumbing:        “define” param matcher

scope:           “scope”
throw:           “throw” expr
escape:          “escape” param “{” expr “}”
compound:        “{” expr “}”
loop:            “loop” “{” expr “}”

if:              “if” “(” expr “)” “{”
                     expr
                 “}” “else” “{”
                     expr
                 “}”

try:             “try” “{”
                     expr
                 “}” (“catch” pattern “{”
                     expr
                 “}”)? (“finally” “{“
                     expr
                 “}”)?
```

Meta-Interpreter Setup

Specification by Meta-Interpreter Enhancement

Defining a computational model that deals with security, upgrade, and debugging in one step is too hard. Rather, we take it in stages. There is a danger of losing security when introducing support for either upgrade or debugging, so we first present -- as the contents of this chapter -- a meta-interpreter for a secure but non-upgradable, non-debuggable E. In Brian Smith terminology, this meta-interpreter reifies *eval* but absorbs *apply* and capability security. (** need to explain this! **) As a result, interpreted subworlds can work transparently with non-interpreted contexts. Indeed, this is the basis for the transparent inter-operation of Java and E objects.

This enables us to define upgrade and debugging support as enhanced meta-interpreters, such that we can design and understand the resulting security properties. A real implementation can then provide the behavioral equivalent of *allowing* (not requiring) code to be run under such an enhanced interpreter. This clearly continues to be a faithful and secure implementation of the semantics specified by the unenhanced interpreter, since one could have run the enhanced interpreter on top of it.

The resulting enhanced interpreter gives us clear answers to the “who is allowed to do-or-see what?” issues needed for debugging securely. Basically, the instantiator of an interpreted subworld holds the only debugging capabilities to that world. For anyone else to have debugging access, they must get it from the interpreter-starter. However, the interpreter-starter can only obtain a debugger’s-eye-view on an object if they also have the object. Not surprisingly, this follows KeyKOS discipline rather well. Ironically, these same debugging hooks also enhance security, by providing us a discretion check enabling confinement. (** explain & refer somewhere **)

Name Spaces

1. *Keywords* are reserved for use by the grammar, and eventually, perhaps by user-defined macros (syntactic extensions). Each keyword is its own Terminal type, and these are not identifiers. Some keywords are reserved for future use.
2. *Nouns* are E's term for variable names, since they are a way to refer to *things*. Every use-occurence of a variable name is a `noun`, and corresponds to a defining-occurence -- or `definer` -- of the same name. The rules for matching `noun` to `definer` are purely static.

   In the kernel language, each use-occurence statically corresponds to exactly one defining-occurence by a simple rule: the scope of a definer extends textually left-to-right from the point of its definition until the corresponding close curly, except where it is shadowed by an inner definition of the same name. However, the definition of *corresponding* is a bit tricky. (E's syntactic sugar usually follows the same rule, with the exception of the for-loop and conditional-or expressions.)
3. *Verbs* are E's terms for method or selector names, since they are a way to request *actions*. The defining-occurence appears in a method definition, and the use-occurence occurs in a call or send expression. The correspondence between the two is many-to-many and not statically determinable.
4. *Behavior-names*. For all the elegance of anonymous closures, to my knowledge no existing systems upgrade old instances of these to behave according to new versions of their code. By contrast, behavior-names in E's object and plumbing expressions enable such upgrades. Behavior names are composed from definers according to the rules in *Upgrading Behaviors*.

   The following meta-circular interpreter ignores upgrade and debugging-support issues. Without these issues,

   ```
   define foo { ... }
   ```

   is equivalent to

   ```
   define foo := define _ { ... }
   ```

   since the remaining significance of `foo` is only as a `definer`.

Meta-Interpreter Skeleton: Evaluation

As with the Lambda Calculus and related languages (most notably Scheme), the heart of computation within an object is the *evaluation* of an *expression* in a lexical *scope*. E's expressions are exactly those kernel parse node types defined by the `expr` production above. In the Lambda Calculus, a scope is an immutable mapping from names (in E, *nouns*) to values. In the non-debuggable, non-upgradable E presented here, a scope is also an immutable mapping from nouns to values, but these values are termed *slots*, since they are expected to hold a variable's value. Slots are similar to *locations* in Scheme's semantics, except that the E programmer can bind whatever object they want to be a noun's slot, and can obtain the slot of an in-scope noun. A primitive mutable slot type is available -- and used by default in the expansion of the sugar language -- enabling classic side effects by assignment.

So altogether, we have four distinct indirections to get from a variable name to the object it designates:

1. **Static Correspondence**
   `expr: noun -> pattern: definer`
   The statically analyzable correspondence between use-occurences of a variable names (noun expressions) and a defining occurence of the same name (a definer pattern). In the kernel language, this strictly follows the left-to-right definer to corresponding close curly rule. The interpreter presented below makes less use of this static analyzability than it probably should, instead managing scopes carefully so that the run-time name/scope uniqueness exactly matches the static correspondence rule.
2. **Lexical Loopup**
   `scope[identifier] -> slot`
   A scope is an immutable mapping from names to slots. Each time an expression is evaluated, a distinct scope is provided, so it will often look up a distinct slot. For example, the equivalent of object's instance variable storage is a scope allocated per-instance that maps instance variable names to slots holding the values. (Note that E has no distinct notion of instance variables. The effect described is an outcome of the semantics presented here.)
3. **Mutable State**
   ```
   slot getValue -> reference   # reading a variable's value
   slot setValue(expr)          # assignment
   ```
   When a variable name is used in a noun expression (as other than the left-side of an assignment), this turns into a slot lookup as described above, followed by asking that slot for its current value. Assignment is turned into a request to the slot to change its value. A `slot`-expression skips step three, and just returns the slot itself as the value of the expression. Variable can *vary* precisely because slots can change their value over time.
4. **Designation**
   `object reference -> object`
   The value returned by step 3 shouldn't be thought of as the object itself, but as a

reference (or pointer, or capability) to the object. When drawing a heap of objects, we typically depict the objects themselves as circles or blobs, and objects references as arrows. If object Alice *points at* object Bob, then Alice holds the tail of an arrow whose head is attached to Bob. Since E is a distibuted language, we give these arrows more semantics than usual: in particular, Alice and Bob can be on different machines, in which case the reference will span machines. Such *DEFERRED* references have a restricted semantics that reflect the inescapable difficulties of distributed computing -- like partial failure. This is documented fully in the *E Reference Mechanics*.

In Lambda Calculus, expression evaluation takes in an expression and a scope, and produces a value. In E, expression evaluation similarly takes in an expression and a scope, but produces an outcome. The three forms of outcome are

1. Success, in which case eval returns a pair of a resulting value and a resulting scope. The resulting scope is a superset of the input scope, but possibly containing further bindings. (** need to specify shadowing rules **)
2. Failure, in which case eval throws an object as the *exception* indicating what problem occured. See *throw* and *try/catch/finally* below.
3. Escape, as documented in the semantics of *escape* below. Both *failure* and *escape* are forms of *non-local exit*.

The following meta-interpreter absorbs the call-return stack discipline, so that returning successfully in the interpreted language is represented by eval returning successfully, and non-local exit in the interpreted language is represented by eval performing a non-local exit.

The natural object-oriented way to define the *eval* function would be to distribute it into a set of *eval* methods defined on each expr-parse-node type. (Indeed, the bootstrap interpreter written in Java does exactly this.) However, this has the wrong extensibility property: we wish to hold the definition of the kernel language fixed over long periods of time, while we also expect to define enhanced semantics for executing E -- especially for debugging and upgrade -- by enhancing this interpreter. Therefore we need to enable multiple interpreters to co-exist in one system, and to allow the others to be defined by incremental modifications to this one. To do this, we write the interpreter as a simple function containing a big switch. Each branch of the switch matches one of the expression types above, and is shown in the corresponding section:

```
define eval(expr, scope) {
    switch (expr) {
        ...
        match e‘...‘ {
            ...
        }
        ...
    }
}
```

Meta-Interpreter Skeleton: Pattern Matching

In Lambda Calculus, defining occurences of nouns only appear in parameter lists. Most programming languages also have a variable declaration/definition construct that can appear in a block of code. Where classically one has defining occurences of variable names, E instead has *patterns*. Where one classically binds a name to an initial value -- by argument matching or initialization -- E instead *pattern matches* the pattern against the initial value. When the pattern is the expansion of a simple Identifier, the effect is identical to the classical case. In other words, the classic parameter variable declarations and local variable definitions are degenerate (but common!) forms of E's pattern matching mechanism.

As with *eval* we define E's pattern matching routine, *testMatch*, with a global function that dispatches (using E's *switch*) on the type of the pattern. The Java implementation of the bootstrap interpreter instead distributes the clauses into the Pattern parse-node types.

```
define testMatch(patt, scope, specimen) {
    switch (patt) {
        ...
        match e‘...‘ {
            ...
        }
        ...
    }
}
```

The *testMatch* function takes in a Pattern parse-tree, a scope as the context in which to perform the match, and a run-time value to match it against. The outcomes:

1. If the match is successful, *testMatch* returns a one-element array containing a scope derived from the imput scope, but also containing any bindings resulting from the match.
2. If the match is unsuccessful, *testMatch* returns `null`.
3. Various components of the match may perform a non-local exit (failure or escape), in which case *testMatch* is so exited.

As you will see below, by returning a one-element array for the success case, rather than just returning the result directly, we enable the use of *matchBind* expressions in the interpreter to both test for success and bind the result in a compact fashion.

Miscellaneous Interpreter Functions

The *mustMatch* function is like *testMatch*, except that an unsuccessful match results in failure rather than returning null, and therefore a successful match can just return the scope directly. If *mustMatch* returns, it will always return a scope.

```
define mustMatch(patt, scope, specimen) {
    define [result] := testMatch(patt, scope, specimen)
    result
}
```

Sometimes we need to introduce names into a scope before we have enough information to bind them to Slots. In this case, we *reserve* the names by binding them to Promises for Slots, and then later resolve the forward reference by forwarding the corresponding Resolvers.

```
define reserve(names, scope) {
    define resolvers := TableEditorImpl()
    for name in names {
        define [prom, res] := Promise()
        scope := scope with(name, prom)
        resolvers[name] := res
    }
    [resolvers snapshot, scope]
}
```

Given a pattern and a scope, *reserve* reserves all names this pattern would bind by returning a pair of a mapping and a new scope. This new scope is derived from the original by binding each name to a promise for a new slot. The mapping maps each of these names to the corresponding Resolver.

```
define resolve(resolvers, scope) {
    for name => res in resolvers {
        res forward(scope[name])
    }
}
```

Should the pattern whose bindings were reserved succeed at matching, *resolve* will resolve the previous reservations to the slots resulting from the successful match.

Defining Behaviors: Method & Matcher

Classically, an object is an encapsulated package of state and behavior, and E's semantics reflects this faithfully.

Objects are defined by the *object* expressions, of which there are two kinds: *methodical* expressions -- for defining *methodical objects*, and *plumbing* expressions, for defining objects that act as message plumbing. An object expression evaluates in a scope to an object that behaves as described by the object expression. For example, the following program in the full E language

```
define Point(x, y) {
    define self {
        to getX {x}
        to getY {y}
    }
}
```

Sample command-line interaction in the context of the above definition:

```
    ? define pt := Point(3, 5)
    ? pt getX
    # value: 3
```

expands into the following Kernel E program:

```
define Point : mutable {
    to run(x : final, y : final) {
        define self : final {
            to getX() {x}
            to getY() {y}
        }
    }
}
    ? define pt : mutable := Point run(3, 5)
    ? pt getX()
    # value: 3
```

In the original program, it seems we've defined *Point* to be a function. In the expansion, we see that *Point* is actually an object with a single *run* method that takes two arguments. After all, kernel-E is a pure object language: all values are objects, and all inter-object interaction (with the exception of equality) is purely by message sending. However, E's sugar streamlines the use of objects to express conventional functional/procedural patterns. *run* is E's default verb. If left out, it will be supplied in the expansion to the kernel language. An apparent function definition actually defines an object with a *run* method.

The behavior of Point's *run* method is to define and return an object (named *self* within the scope of the *run* method, and therefore named *self* to itself). By *behavior*, we mean this is what Point does in response to a *run* message of two arguments. This returned object's behavior is similarly described by two methods: getX and getY. (Another tasty bit of sugar is that zero-argument argument lists may be left out of both definition and call. However, you may not leave out both the verb and the argument list.)

This returned object acts like a conventional point object, but where does the state of the object come from? Somehow, *x* and *y* are acting like its classical instance variables, but there doesn't seem to be anything special about their declaration. Indeed there isn't. The scope of *x* and *y* extends from their definition until the close curly at the end of the *run* method. The *self* object expression within this scope can therefore refer to these slots by using their names.

So the *state-nouns* of an object expression are those nouns used within the object expression that *statically correspond* (see above) to noun definitions outside the object expression. An object expression evaluates in a scope to an object. The object's *state* is the subset of this scope containing the object expression's state-nouns. When an object receives a message, it executes a corresponding method or matcher in a scope which is a child of this state.

Method Definition

```
method:         "to" verb "(" pattern* ")" "{"
                    expr
                "}"
```

Method definitions only occur inside object expressions, and describe how the resulting object responds to a message with the same verb and arity.

When an instance of an object expression receives a message whose name and arity match one of its methods, that method is invoked in a child of the state captured when the object was instantiated (when the object-expression was evaluated). The sequence of arguments is first matched against the sequence of parameters. If this doesn't succeed, an exception is thrown (even if the optional matcher has been provided). If this does succeed, the body expr is then evaluated in a child of the resulting scope, and the outcome of expr's evaluation is the outcome of the invocation.

Evaluation Rule:

```
define doMethod(method, state, args) {
    scope := state sprout
    define patterns := method patterns
    assert(args length == patterns length)
    for i in 0 till(args length) {
        scope := mustMatch(patterns[i], scope, args[i])
    }
    eval(method bodyExpr, scope)[0]
}
```

Matcher Definition

```
matcher:    "match" pattern "{"
                expr
            "}"
```

A matcher describes an object's "behavior of last resort", ie, how it should respond to a message when it has no method of the same verb and arity. In this case, the matcher is invoked in a child of the lexical scope captured on instantiating the object. The message is turned into a pair (a two-element sequence) and matched against the pattern. If this fails an exception is thrown. If this succeeds, the body expr is evaluated in a child of the scope resulting from the match, and the outcome is the outcome of the invocation.

Evaluation Rule:

```
define doMatcher(matcher, state, message) {
    scope := state sprout
    scope := mustMatch(matcher pattern, scope, message)
    eval(matcher bodyExpr, scope)[0]
}
```

Params & Patterns

Params are Patterns that define at most one variable.

The Definer Param

```
definer:        Identifier “:” expr
```

Defines a new noun with the given name. The scope of this noun lasts left-to-right from the `definer` until the corresponding close curly, except as shadowed by inner blocks that define the same name. If we ignore for a moment what happens when *expr* uses the noun it is helping to define, we have the following simpler explanation:

To match a *definer* in a scope to a specimen, first evaluate expr in that scope. If successful, this results in a returned value and a new descendent scope. The returned value is assumed to be a *SlotMaker.* A SlotMaker is an object that responds to *makeSlot(initialValue)* by returning a *Slot.* A Slot is an object that responds at least to *getValue* (no arguments), and, if it represents a *MutableSlot*, to *setValue(newValue)*. So we bind the *Identifier* to the result of sending *makeSlot(specimen)* to the presumed SlotMaker. The scope resulting from the definer is the original scope enhanced both by this binding of *Identifier*, and by any bindings produced by *expr*.

The true explanation differs in the following ways:

Since *expr* is to the right of *Identifier*, it is within *Identifier*'s scope. However, expr must be evaluated before we can create the slot that the name designates. So the name starts off bound to a promise for the slot. Within this new scope, *expr* is evaluated, potentially adding yet further bindings. Assuming expr evaluates successfully, we proceed to make a presumed Slot as above. This object then becomes the resolution of the promise.

testMatch Rule:

```
   match epatt‘@name : @expr‘ {
      define [resolvers, scope2] := reserve([name], scope)
      define [slotMaker, scope3] := eval(expr, scope2)
```

```
        define slot := slotMaker makeSlot(specimen)
        resolve(resolvers, scope with(name, slot))
        [scope3]
    }
```

There are several well known SlotMakers, and compilers and discretion-checkers will often have special knowledge of some of these. For example:

```
define mutable makeSlot(value) {
    define slot {
        to getValue {value}
        to setValue(newValue) {
            value := newValue
        }
    }
}

define final makeSlot(value) {
    define slot getValue {value}
}
```

The *mutable* SlotMaker makes a Slot that acts like our normal notion of an untyped assignable (ie, *mutable*) variable, whereas the *final* SlotMaker makes a slot that may be read but not changed.

By convention, a Type is a SlotMaker. In addition, a Type has a *vouch* method for vouching for instances of the Type, and a *final* method to make SlotMaker that makes final Slots. To E, all Java classes are Types, and respond according to the following three methods:

```
    to vouch(specimen) {
        # a small bit of implicit coercion magic can happen here.
        # See The E-to-Java Binding Specification
        if (self isInstance(specimen)) {
            specimen
        } else {
            throw ... //some appropriate exception
        }
    }
    to makeSlot(specimen) {
        define value := self vouch(specimen)
        define slot {
            to getValue {value}
            to setValue(newValue) {
                value := self vouch(newValue)
            }
        }
    }
    to final {
        define typedFinal makeSlot(specimen) {
            define value := self vouch(specimen)
            define slot getValue {value}
        }
    }
```

This allows types to constrain the possible values of variables in ways familiar from statically typed languages, and in ways that enable similar compiler optimizations, but all in term of a purely run-time semantics. For example:

```
    define Character : final := load:java.lang.Character
    ...
    define foo : Character := ...
    ...
    foo := c
    ...
```

Both human and compiler know that *foo* can only come to hold instances of Character. If *c* isn't a Character, the assignment will fail (with a thrown exception) and *foo* will be unaffected. By special dispensation, when the compiler can show that a binding or assignment would always fail (even if it can't show that it would ever be executed), it may reject the program with a compile-time error.

So, for example, if we define a subrange type:

```
define subrange(start, bound) {
    define self {
        to vouch(specimen : Number final) {
            if (start <= specimen && specimen < bound) {
                specimen
            } else {
                throw ... //some appropriate exception
            }
        }
        to makeSlot(specimen) {
            ...  //as shown in Java class behavior above
        }
        to final {
            ...  //as shown in Java class behavior above
        }
    }
}
```

(An actual system would not have so much repetition of code patterns, but avoid it well involves the *inheritance* pattern, which should not be introduced so soon.)

If we then declare

```
    ...  x : subrange(3, 5)  ...
```

we know that in each resulting scope, x will be bound to a mutable slot that only holds values between 3 (inclusive) and 5 (exclusive). On the other hand, if we declare

```
    ...  x : subrange(3, 5) final  ...
```

then in each scope x will be bound to final slot whose initial and permanent value will be in this range.

Does `subrange`'s `vouch` need to ask `Number` to `vouch` for `specimen`? In order for `subrange` to vouch for `specimen`, it has to know that `specimen` will *always* act like an integer within the specified range. Integers are just objects responding to a protocol. An object acting like an integer may say it's less than 5 one moment, and act like 6 another moment, so subrange itself needs assurance that it can trust the consistency of `specimen`'s behavior.

Finally, we can use the SlotMaker *defineSlot* to use the specimen itself as a Slot:

```
define defineSlot makeSlot(specimen) { specimen }
```

The Ignore Param

```
ignore:         "_"
```

Matches any specimen, but binds nothing. Most commonly used for a parameter in a method declaration when the method has no need for the corresponding argument.

testMatch Rule:

```
match epatt‘_‘ {
    [scope]
}
```

The SuchThat Pattern

```
suchThat:       pattern “?” expr
```

A suchThat pattern matches a specimen iff the contained pattern matches the specimen, and in the context of the resulting bindings, the contained expr evaluates to true. If the contained pattern doesn’t match, the contained expression isn’t evaluated.

testMatch Rule:

```
match epatt‘@patt ? @expr‘ {
    if (testMatch(patt, scope, specimen) =~ [scope2]) {
        define [ok, scope3] := eval(expr, scope2)
        if (ok) { [scope3] }
    }
}
```

Although it’s arguably bad style, the above piece of code relies on the property that an *if-then* acts like (indeed, expands to) an *if-then-else* in which the automatically supplied *else* evaluates to `null`.

This form is especially useful when you want to express pre-conditions on the call, rather than constraints on the variables. (** need examples **)

The Tuple Pattern

```
tuple:          “[” pattern* “]” “+” pattern
```

Matches a tuple iff

1. It has at least as many elements at the number of patterns between the brackets.
2. Each such pattern matches the corresponding tuple element
3. The pattern after the “+” matches the remainder of the tuple.

Execution proceeds left-to-right, and scope accumulates left-to-right, so each match occurs in the context of all matches to its left. A failing match prevents any matches to its right from being attempted.

testMatch Rule:

```
match epatt‘[@patts...] + @restPatt‘ {
    define length := patts length
    escape return {
        if (length > specimen length) {
            return(null)
        }
        for i in 0 till(length) {
            if (testMatch(patts[i],scope, specimen[i]) =~ [s2]){
                scope := s2
            } else {
                return(null)
            }
        }
    }
    testMatch(restPatt, scope, specimen slice(length))
}
```

This allows patterns to extract values from nested tuple structures in ways familiar to Prolog programmers. A silly example:

```
define car([x] + _) {x}
define cdr([_] + y) {y}
```

In practice, the main use of this is to obtain mutiple resturn values from a call, such as our own `eval` function. Above, when we said

```
define [ok, scope2] := eval(expr, scope)
```

this expands to

```
define [ok, scope2] + _x ? _x == [] := eval(expr, scope)
```

(all expansion-created temporary names begin with an underscore)

On the right, `eval` was returning a tuple (immutable array) of two elements. On the left, the zero'th element of this array is matched against the zero'th pattern, binding the noun `ok`, and the one'th element is matched against the one'th pattern, binding `scope2`. The remaining part of the tuple is matched against "`_x ? _x == []`" which only succeeds if the remaining part of the tuple is the empty tuple, ie, if the original tuple was exactly two elements long.

Expressions

The Literal Expression

```
expr:       BigIntegerLiteral | DoubleLiteral
|           StringLiteral | CharLiteral
```

This simply evaluates to the described value. The syntax is as in Java, including full Unicode support, but without a precision limit on integers.

Evaluation Rule:

```
   match x ? x isa(LiteralExpr) {
      [x getValue, scope]
   }
```

Examples:

```
   35
   3.14159
   “Hello world\n”
   ‘\\’
```

The Noun Expression

```
expr:       noun
```

Evaluates to the current value of the variable, according to the Slot. If the name is bound only to a DEFERRED reference to a Slot (typically a Promise for a not-yet-resolved forward definition), then the Noun expression evaluates to a Promise for an eventual value of the eventual Slot. (** The ordering weakness seems dangerous. Should it simply error instead? **)

Evaluation Rule:

```
match noun ? (noun isa(NounExpr)) {
    define getValue(slot) {
        if (E state(slot) == “DEFERRED”) {
            slot <- getValue
        } else {
            slot getValue
        }
    }
    [getValue(scope[noun name]), scope]
}
```

Rather than being in Kernel E, the noun production could have been defined by expansion, as the noun

```
foo
```

is equivalent to the expression

```
(&foo) getValue
```

The Slot Expression

```
slot:       “&” noun
```

Evaluates to the slot which holds the value of the variable.

Evaluation Rule:

```
match e‘&@noun‘ {
    [scope[noun name], scope]
}
```

This is used in the expansion of the conditional-and (&&) and conditional-or (||) sweeteners, but its most intriguing use is for one-way eventual constraint expressions:

```
define x : observableSlot := 3
define y : observableSlot := 5
define z : constrain := ever(&x) + ever(&y)
```

With the appropriate definitions of `observableSlot`, *constrain*, and *ever*, we can read this as "*constrain z to always eventually be the sum of the latest values of x and y*". See *Lamport Slots and Constraint Programming*.

The Assignment Expression

```
expr:      noun ":=" expr
```

Changes the value of the variable to the new value. More precisely, asks the Slot for this noun in the current scope to change its current value to the value of the expression.

If the variable was declared `final`, an implementation may (**must??**) reject this statically rather than dynamically.

Evaluation Rule:

```
match e‘@leftNoun := @rightExpr‘ {
    define slot := scope[leftNoun name]
    define [result, scope2] := eval(rightExpr, scope)
    [slot setValue(result), scope2]
}
```

An example of multiple facets sharing common mutable slots:

```
define readWritePair(value) {
   [define reader() { value },
    define writer(newValue) { value := newValue } ]
}
```

readWritePair returns two function-objects that share common state: the slot named "value". Each time readWritePair is called, it returns a new pair of functions that share a new Slot:

```
? define [r1,w1] := readWritePair("foo")
? define [r2,w2] := readWritePair("foo")
? w1(“bar”)
? w2(“baz”)
? r1()
# value: bar
? r2()
# value: baz
```

Since the two objects in a pair provide different powers over shared state -- one can only read and the other can only write -- we often refer to them of two *facets* of composite object defined by this state and all the behaviors that can operate on it. Although it is often a useful way to speak, only the individual facets are actual objects as far as E is concerned. This technique demonstrates that the power traditionally associated with *capability bits* -- to have different references to an object convey different authorities -- is easily achived with nothing but pure objects. Different authority over common state is represented by access to different objects that share that state. All references to the same object grant equal authority: the authority to invoke any of its public methods.

In some sense, this is all implied by the early definitions of objects: Classically, we say an object is a combination of state and behavior. Well, before objects there were modules, which were a combination of file-scope global variables and procedures that had them in scope. (Remember C’s file-static variables?) In going from modules to classes, we changed the data a module could access to be multiply instantiable. In other words, different instances of the same class attach the same code with different data. The facet technique above shows the power of also attaching different code to the same data.

The Sequence Expression

```
expr:       expr “;” expr
```

Evaluates these in order. The resulting value is the value of the right-hand expression.

Evaluation Rule:

```
match e‘@leftExpr ; @rightExpr‘ {
    define [_, scope2] := eval(leftExpr, scope)
```

```
        eval(rightExpr, scope2)
    }
```

The Call Expression

```
expr:       expr verb “(” expr* “)”
```

First, the receiver expression and the argument expressions are evaluated in left-to-right order. The receiver is then called synchronously with a message

1. whose verb is the stated verb
2. whose args are a tuple of the evaluation-results of the arg expressions
3. and whose continuation is the outcome of the call.

This is conventional *do-it-now* call-return ordering, where the receiver returns before the caller continues. Since the caller is blocked until the receiver returns, the receiver must be in the same vat as the caller, ie, the receiver expression must evaluate to a KEPT reference. Since both caller and receiver are in the same Vat, any side effects made by the receiver to shared synchronous state must be seen by the caller after the call.

Although it’s not apparent in the current meta-interpreter, two forms of outcome can be reported to the continuation: success -- reported by forwarding the continuation to the resulting value, and exceptional -- reported by smashing the continuation with the explanatory problem. (** update terminology and account for escape **)

Evaluation Rule:

```
    match e‘@receiverExpr @verb (@argExprs...)‘ {
        define [rec, scope2] := eval(receiverExpr, scope)
        define args := []
        for argExpr in argExprs {
            define [arg, scope3] := eval(argExpr, scope2)
            args += [arg]
            scope2 := scope3
        }
        [E call(rec, verb name, args), scope2]
    }
```

The Send Expression

```
expr:      expr "<-" verb "(" expr* ")"
```

First, the receiver expression and the argument expressions are evaluated in left-to-right order. A new Promise is created, and we queue onto the receiver's vat a pending delivery to the receiver of a message

1. whose verb is the stated verb
2. whose args are a tuple of the evaluation-results of the arg expressions
3. and whose continuation is the new Promise's resolver.

The value of the send expression is immediately the acceptor of the new Promise, and the sender immediately continues. This is E's optimistic event-loop scheduling primitive, where the sender continues without blocking, and the receiver receives the message in a brand new event. Any side effect performed by the receiver cannot affect the sender's event. (*Chronological encapsulation*)

Although it's not apparent in the current meta-interpreter, two forms of outcome can be reported to the continuation: success -- reported by forwarding the resolver to the resulting value, and exceptional -- reported by smashing the resolver with the explanatory problem. (** update terminology and account for escape **)

Evaluation Rule:

```
match e‘@receiverExpr <- @verb (@argExprs...)‘ {
    define [rec, scope2] := eval(receiverExpr, scope)
    define args := []
    for argExpr in argExprs {
        define [arg, scope3] := eval(argExpr, scope2)
        args += [arg]
        scope2 := scope3
    }
    [E send(rec, verb name, args), scope2]
}
```

The MatchBind Expression

```
expr:       expr “=~” pattern
```

The expression is evaluated, and then matched against the pattern. The value is a boolean reporting the outcome of the match.

Evaluation Rule:

```
match e‘@leftExpr =~ @patt‘ {
    define [specimen, scope2] = eval(leftExpr, scope)
    if (testMatch(patt, scope2, specimen) =~ [scope3]) {
        [true, scope3]
    } else {
        for name in expr namesOut {
            scope := scope withBrokenBinding(name)
        }
        [false, scope]
    }
}
```

The Definition Expression

```
expr:       “define” pattern “:=” expr
```

We first reserve all the names defined by the pattern. Then we evalate the expression in the resulting scope, producing the specimen. We then match this with the pattern. If this fails, we throw an exception. If this succeeds, the bindings produced by the match become the resolution of our reserved names, and the value of the define expression is the specimen.

The reservation mechanism is to initially bind each name to a new Promise acceptor, and then to resolve the reservation by forwarding the corresponding resolver. For you Lispers, this is a `letrec` than works. Besides capturing such names on the right side in object expressions, one may also asynchronously send it messages, and store it in data structures. The cleanliness of the E inheritance pattern depends on the cleanliness of this definition.

Evaluation Rule:

```
match e‘define @patt := @rightExpr‘ {
    define [rslvs, scope2] := reserve(patt namesOut, scope)
    define [specimen, scope3] := eval(rightExpr, scope2)
    resolve(rslvs, mustMatch(patt, scope, specimen))
    [specimen, scope3]
}
```

The Methodical Expression

```
methodical:     “define” param “{”
                    method*
                    matcher?
                “}”
```

An object expression evaluates to an object which is an *instance* of the object expression, and this particular case of evaluation is also called *instantiation*. Debugging issues aside, the semantics of such an instance is fully described by four attributes.

1. The object expression describes the behavior of its instance.
2. The object expression will often contain use-occurences of nouns (variable names) whose defining occurence is outside the object-expression. For each such name, the resulting variable in the instantiating context is captured by the instance. The state of the instance consists of such variables.
3. Each instantiation endows the instance with a unique EQ-identity. E’s equality tests will distinguish otherwise-equivalent instances on this basis.
4. Vat-host. E is a distributed language, so its semantics makes explicit that each instance is somewhere. E’s notion of a place for an object to be is a *Vat*. Each Vat hosts a number of objects, and all objects are hosted by exactly one Vat.

On receiving a message, an instance will first attempt to find a matching explicit method (a method appearing in the object expression). Failing that, it will attempt to find a matching Miranda method -- a set of standard methods provided for those objects that don’t provide overrides of their own. (This primitive form of inheritance in the only kind directly supported by the E kernel.) Failing that, the message is given to the matcher if any. If there isn't any, the Miranda matcher is to throw a NoSuchMethodException, describing the message that couldn't dispatch.

Evaluation Rule:

```
match e‘define @param { @methods... @optMatcher }‘ {
    define [rslvs, scope2] := reserve(param, scope)
    define state := ... //subset of scope2 used freely in the behavior
    define obj match [verb, args] {
        escape return {
            for meth in methods + mirandas {
                if (verb == meth verb ...
                     && args length == meth arity) {

                    return(doMethod(meth, state, args))
                }
            }
            if (optMatcher != null) {
                doMatch(optMatcher, state, [verb, args])
            } else {
                throw ... //some appropriate exception
            }
        }
    }
    resolve(rslvs, mustMatch(param, scope, obj))
    [obj, scope2]
}
```

The Plumbing Expression

`expr:` “**`define`**” `param matcher`

A plumbing expression is much like a methodical expression with no methods, except that no Miranda methods are provided either. This allows one to define “message plumbing” -- objects that forward messages generically without dispatching on them.

Evaluation Rule:

```
match e‘define @param @m‘ ? (m isa(Matcher)) {
   define [rslvs, scope2] := reserve(param, scope)
   define state := ... //subset of scope2 used freely in m
   define obj match [verb, args] {
      doMatch(m, state, [verb, args])
   }
   resolve(rslvs, mustMatch(param, scope, obj))
   [obj, scope2]
}
```

The Scope Expression

```
expr:       “scope”
```

Reifies the current lexical environment. This environment should contain slots for exactly those variables whose definers are in scope, and, if they are defined outside the current object or plumbing expression, are used freely by that expression.

Evaluation Rule:

```
match e‘scope‘ {
   [scope, scope]
}
```

The Throw Expression

```
expr:       “throw” expr
```

Evaluates expr and smashes the continuation with the resulting problem. Evaluation terminates outward through enclosing `try` blocks, running their `finally` clauses, until a matching `catch`-clause is found (or unless an intervening `finally` clause hijacks flow of control by initiating its own non-local termination). Should evaluation terminate all the out to the event loop, the resolver of this event's initiating message is smashed with the problem.

Evaluation Rule:

```
match e‘throw @problemExpr‘ {
    define [problem, _] := eval(problemExpr, scope)
    throw problem
}
```

The Escape Expression

```
expr:        “escape” param “{”
                  expr
             “}”
```

First, in a child of the current scope, *param* is bound to an *escapeHatch* for exiting the escape expression, and then expr is evaluated in the resulting scope. If the escapeHatch is not invoked, expr completes normally, and its value is the value of the escape expression. If the escapeHatch is called with a `run(val)` during the evaluation of expr, expr terminates outward from that point (running intervening `finally` clauses), until the escape expression is reached. The escape expression's value is then the argument to `run()`. As a convenience, an escapeHatch also has a no-argument `run()` method that is equivalent to `run(null)`.

Once an escape expression has completed, its escapeHatch is disabled.

Evaluation Rule:

```
match e‘escape @param { @bodyExpr }‘ {
    escape hatch {
        scope := mustMatch(param, hatch, scope sprout)
        define [val, _] := eval(expr, scope)
        [val, scope]
    }
}
```

The Compound Expression

```
expr:        "{"
                 expr
             "}"
```

Provides a nested scope, so that definers in expr don't introduce names into the enclosing conext.

Evaluation Rule:

```
match e‘{ @bodyExpr }‘ {
    define [val, _] := eval(bodyExpr, scope sprout)
    [val, scope]
}
```

The Loop Expression

```
expr:        "loop" "{"
                 expr
             "}"
```

Evaluates expr repeatedly, each time in a new child of the current scope. The loop can only be exited by throws or escape hatches.

Evaluation Rule:

```
match e‘loop { @bodyExpr }‘ {
    loop {
        eval(bodyExpr, scope sprout)
    }
}
```

The If Expression

```
expr:      “if” “(” expr “)” “{”
               expr
           “}” “else” “{”
               expr
           “}”
```

In a child of the current scope, evaluates the condition to a boolean. If true, evaluates the then expression in the context of the resulting bindings, and the result is the value of the if expression. If false, evaluates the else expression in a new child of the original scope, and the result is the value of the if expression.

Evaluation Rule:

```
match e‘if (@cond) {@then} else {@els}‘ {
    define [ok, scope2] := eval(cond, scope sprout)
    define [val, _] := if (ok) {
        eval(then, scope2)
    } else {
        eval(els, scope sprout)
    }
    [val, scope]
}
```

The Try-Catch-Finally Expression

```
expr:      “try” “{”
               expr
           “}” (“catch” pattern “{”
               expr
           “}”)? (“finally” “{“
               expr
           “}”)?
```

In a child of the current scope, evaluate the first expr. If it completes normally, in a new child of the original scope, evaluate the `finally` expr if any, and the outcome of `try` expr is the outcome of the first expr, unless hijacked by the `finally` expression.

If expr terminates by throwing (ie, by smashing its continuation), in a new child of the original scope attempt to match the problem with the `catch` clause's pattern. If it matches, run the `catch` expression in the context of the resulting bindings. Then evaluate the `finally` clause as above. The outcome of the `try` expression is the outcome of the `catch` expression. If the pattern does not match, proceed as if there was no catch clause.

Evaluation Rules:

```
match e‘try {
            @shot
         } catch @patt {
            @mitt
         } finally {
            @fin
         }‘ {

    define s2 := scope sprout
    define [val, _] := try {
       eval(shot, s2)
    } catch ex ? (testMatch(patt, s2, ex) =~ [s3]) {
       eval(mitt, s3)
    } finally {
       eval(fin, s2)
    }
    //if we get here
    [val, scope]
}
```

CHAPTER 2

The E Language Grammar

(** An actual implementation may choose to expand differently or not at all, as long as there is no observable difference. In particular, user-visible parse tree produced by the `e‘...‘` form must produce parse trees according to the canonical expansion. **)

CHAPTER 3 ELang Utilities

Scope Building Blocks

Scope

Slot

FinalSlot

LamportSlot

CHAPTER 4 *Upgrading Behaviors*

are defined by the Object-Definition and Plumbing-Definition expressions below. When their initial param has a `definer`, this name serves two purposes. (**This needs its own separate section somewhere. **)

1) The `definer` is the defining occurence of a noun. At runtime, the definition evaluates to an object, which becomes the initial value of a variable of the `definer's` name. This conventional use only employs the noun name-space.

2) The object produced by case #1 is a combination of two ingredients: behavior and state. The behavior is statically described by the code of the defining expression. All objects produced by evaluating such defining expressions are *instances* of this expression, differing from each other only in their state (captured from their lexical context at evaluation time). For both upgrade-for-prototyping and upgrade-for-release, we need to be able to change the code that already instantiated objects obey.

If the initial param has a `definer`, and when the initial `param` of each such lexically enclosing definitions is also has a `definer`, the path of these names is the Behavior-name. A development context that maintains an association of Behavior-names to behaviors can upgrade old behaviors in place to reflect newly accepted code. If the instances designate their behavior by reference to these same mutable behavior objects, then their behavior is upgraded as a result.