# Eslang Definitions

Eslang is a term tree representation of E code. It is an implementation form, in which many elements of kernel E are already compiled. Specifically:

- Return guards and value guards are compiled in and compiled away where possible
- Some constant elimination is done
- Lots of the redundancy in the direct expansion for E is eliminated
- Object expressions are flattened. Nesting scopes are encoded explicitly, including separation of universals, outer scope, and free variables. Object expressions are replaced with ‘make’ expressions that explicitly instantiate a class object that essentially directly maps to a script object..
- Pattern matching is explicitly coded (as part of the guard expansion).

The first argument of all constructs is a placeholder (called ‘loc’) for meta-information, especially source positions. Names between single quotes simply name the previous construct (for use in the action code associated with the rules).

*Literals from the scanner*

```
string      :   <STRINGLITERAL>;
literal     :   <INTEGERLITERAL>
            |   <FLOATINGPOINTLITERAL>
            |   <CHARACTERLITERAL>
            |   string ;
```

*Different single word tokens*

```
name        :   string 'nm' ;
selector    :   string 'sel' ;
```

*A place holder for source locations and meta information. Currently an integer to be able to easily determine which clause in the translator produced a broken eslang expression.*

```
loc         :   <INTEGERLITERAL> ;
```

*The top-leve construct, containing a list of classes and a method for the module behavior to invoke.*

```
module      :   "pkg" "(" loc string ‘packageName’class* 'classes' method 'moduleMethod' ")" ;
```

*All object definitions have been replaced with a top-level class definition (with the correct fully-qualified name) and a ‘make’ expression replacing the nested object expression to instantiate an instance of that definition. All free variables in the object expression (except universals) are passed as arguments in the ‘make’ expression. Outers (top level defines and the Outer scope) that are not universals are always maintained in a single outer scope object and looked up by name.*

```
class       :   "class"  "(" loc name 'className'  ivars  "methods" methods 'ms' ")" ;
```

*This is the make expression that replaces object expressions with an explicit instantiation. The arguments are the free variables from the object expression (except universals).*

```
make        :   "make" "(" loc name 'className' expr* 'args' ")" ;
```

*Different defining occurrence of variables. These are all the same in eslang, but occur in different contexts, and so have different constructs here so we can hang different code off of them.*

```
arg         :   "param" "(" loc string 'name' ")" ;
blockArg    :   "param" "(" loc string 'name' ")" ;
temp        :   "param" "(" loc string 'name' ")" ;
param       :   "param" "(" loc string 'name' ")" ;
```

*These are helpers for lists of the different kinds of variables. The leading token for ‘ivars’ is here; for others, it is in the callers.*

```
ivars       :   "ivars" params 'iv' ;
args        :   |     "(" arg* 'params'  ")"  ;
temps       :   |     "(" temp* 'params' ")"   ;
params      :   |     "(" param* 'params' ")" ;
```

*A references to a non-outer variable. Outer variables are identified during translation, and managed differently.*
ref : "ref" "(" loc name 'nm' ")" ;

*An outer variable reference. The optional base is an outer scope frame. For universal variables, 'base' is absent because they are always directly present from the interpreter. If base is present, then the containing object must have gotten base from it's creator. If base is present, then the variable is looked up in base.*
outer : "outer" "(" loc name 'name' expr? 'base' ")" ;

*A literal value.*
lit : "lit" "(" loc literal 'lit' ")" ;

*Methods on a class. A class might not have any methods.*
methods : | "(" method* 'all' ")" ;

*A method can either be a normal method or the delegation to a superclass. Pattern match methods simply explicitly implement Smalltalk's "messageNotUnderstood". They should probably get their own construct to be less Smalltalk-specific.*
method : "method" "(" loc selector 'selector' "params" args 'args' "locals" temps 'temps' seq 'bd' ")"
| "delegate" "(" loc ref 'sup' ")" ;

*Most of the places in which a sequence of expressions is allowed, so is a single expression, and vice-versa.*
singleExpr : ( call | send | escape | match | assign | ref | outer | if | make | lit | try | finally | meta ) ;
expr : ( singleExpr | seqExpr ) ;
seq : "seq" "(" loc expr* 'subs' ")"
| singleExpr 'sub' ;
seqExpr : "seq" "(" loc expr* 'subs' ")" ;

*These are all the corresponding E constructs.*
call : "call" "(" loc expr 'rcvr' selector 'sel' expr* 'args' ")" ;
send : "send" "(" loc expr 'rcvr' selector 'sel' expr* 'args' ")" ;
assign : "assign" "(" loc ref 'target' expr 'rVal' ")" ;
escape : "escape" "(" loc blockArg 'hatch' seq 'body' ")" ;
match : "match" "(" loc blockArg 'hatch' seq 'breaker' seq 'body' ")" ;
if : "if" "(" loc expr 'cond' seq 'then' seq 'else' ")" ;
try : "try" "(" loc seq 'attempt' blockArg 'patt' seq 'catcher' ")" ;
finally : "finally" "(" loc seq 'attempt' seq 'after' ")" ;

*Only meta-context is supported here.*
meta : "meta" "(" loc "context" ")" ;