$$
\begin{array}{lcl}
e & = & b \mid x \mid \text{let } x = e \ \text{ in } e \mid x \coloneqq e \mid e(e) \mid \text{fun}(x)\{e\} \mid e \text{ op } e \mid \text{toAST}(e) \mid \text{compile}(a) \\
v & = & b \mid \langle \Gamma, x, e \rangle \mid a \\
a & = & [\text{fn}\, x\, a] \mid [\text{base } b] \mid [\text{var } \ \langle \Gamma, x, e \rangle] \mid [\text{op}\, a\, a]
\end{array}
$$

Figure 1: Syntax of our version of Lua Core, extended with constructs to specify Lua2AST

In this section, we specify below the behavior of toAST() and compile() by using the formalization of a subset of Lua semantics, presented in [1] as Lua Core. We use the same formal framework of that work in order to properly compare and contrast our approach for multi-stage programming to that employed by Terra.

Lua Core depicts the notions of lexical scoping, closures and side-effects present in Lua, and is therefore mostly sufficient for our purposes. We extend this specification with a general "binary operator" expression, mimicking Lua operators supported by Lua2AST. This way, we have a recursive rule through which we can model Lua expressions as trees, to be later converted to ASTs. We also include toAST() and compile() as core language operations so we can specify their semantics separately from plain functions.

The syntax of our version of Lua Core is presented in Figure 1. Lua expressions (e) can be base values (b), variables (x), a scoped variable definition (let $x = e\ \text{ in } e$, with $e; e$ as sugar for let $\_ = e\ \text{ in } e$), a variable assignment ($x \coloneqq e$), an application ($e(e)$), a function definition ($\text{fun}(x)\{e\}$) or an operation on expressions (e op e, with semantics defined by a function Op). We extend this by adding operations *toAST(e)* and *compile(a)*. Lua values (v) can be base values (b), closures ($\langle \Gamma, x, e \rangle$) or Lua ASTs ($a$). A Lua AST for a function consists of a root node ($[\text{fn}\, x\, a]$) which may contain nodes that wrap base values ($[\text{base } b]$), variables ($[\text{var } \ \langle \Gamma, x, e \rangle]$) and operations ($[\text{op}\, a\, a]$).

The rules for evaluating Lua expressions over an environment Σ, which is a tuple (Γ, S) containing a namespace $\Gamma : x \to p$ and a store $S : p \to v$ (where p are memory positions) are given in Figure 2. We use $\to$ instead of $\xrightarrow{L}$ as in [1]; where rules have the same names, they have the same semantics as those presented in that work.

The rules for decompiling Lua expressions ($\xrightarrow{D}$) over an environment Σ are the following TODO

Note that $\xrightarrow{D}$ is defined only for variables, base values and the operator, mirroring the implementation of LuaToAST.

Finally, the rules for compiling Lua ASTs ($\xrightarrow{C}$) over an environment Σ are TODO

The evaluation of variables x happens only at compilation time, as evidenced by the evaluation of e_1 using $\to$ in rule CVar.

$$v, \Sigma \to v, \Sigma \quad (\textsc{LVal})$$

$$\frac{\Sigma = (\Gamma, S)}{x, \Sigma \to S(\Gamma(x)), \Sigma} \quad (\textsc{LVar})$$

$$\frac{\begin{array}{c} e_1, \Sigma_1 \to v_1, (\Gamma_2, S_2) \\ p \text{ fresh} \\ e_2, (\Gamma_2[x \leftarrow p], S_2[p \leftarrow v_1]) \to v_2, (\Gamma_3, S_3) \end{array}}{\text{let } x = e_1 \ \text{ in } e_2, \Sigma_1 \to v_2, (\Gamma_2, S_3)} \quad (\textsc{LLet})$$

$$\frac{\begin{array}{c} e_1, \Sigma_1 \to \langle \Gamma_1, x, e_3 \rangle, \Sigma_2 \\ e_2, \Sigma_2 \to v_1, (\Gamma_3, S_3) \\ p \ \text{ fresh} \\ e_3, (\Gamma_1[x \leftarrow p], S_3[p \leftarrow v_1]) \to v_2, (\Gamma_4, S_4) \end{array}}{e_1(e_2), \Sigma_1 \to v_2, (\Gamma_3, S_4)} \quad (\textsc{LApp})$$

$$\frac{e_1, \Sigma_1 \to v_1, (\Gamma, S) \qquad \Gamma(x) = p}{x := e, \Sigma \to v, (\Gamma, S[p \leftarrow v])} \quad (\textsc{LAsn})$$

$$\frac{\Sigma = (\Gamma, S)}{\text{fun}(x)\{e\}, \Sigma \to \langle \Gamma, x, e \rangle, \Sigma} \quad (\textsc{LFun})$$

$$\frac{\begin{array}{c} e_1, \Sigma_1 \to v_1, \Sigma_2 \qquad e_2, \Sigma_2 \to v_2, \Sigma_3 \\ v_3 = Op(v_1, v_2) \end{array}}{e_1 \ \text{ op } e_2, \Sigma_1 \to v_3, \Sigma_3} \quad (\textsc{LOp})$$

$$\frac{e_1, \Sigma \to \langle \Gamma, x, e_2 \rangle, \Sigma \qquad \langle \Gamma, x, e_2 \rangle, \Sigma \stackrel{D}{\to} a}{\text{toAST}(e_1), \Sigma \to a, \Sigma} \quad (\textsc{LAst})$$

$$\frac{\Sigma = (\Gamma, S) \qquad a, \Sigma \stackrel{C}{\to} e}{\text{compile}(a), \Sigma \to e, \Sigma} \quad (\textsc{LComp})$$

$$b, \Sigma \stackrel{D}{\to} [\text{base } b] \quad (\textsc{DBase})$$

$$\frac{\Sigma = (\Gamma, S, F)}{x, \Sigma \stackrel{D}{\to} [\text{var } \ \langle \Gamma, \_, x \rangle]} \quad (\textsc{DVar})$$

$$\frac{e_1, \Sigma \stackrel{D}{\to} a_1 \qquad e_2, \Sigma \stackrel{D}{\to} a_2}{e_1 \ \text{op} \ e_2, \Sigma \stackrel{D}{\to} [\text{op}\, a_1\, a_2]} \quad (\textsc{DOp})$$

$$\frac{e, \Sigma \stackrel{D}{\to} a}{\langle \Gamma, x, e \rangle, \Sigma \stackrel{D}{\to} [\text{fn } x\, a]} \quad (\textsc{DFn})$$

$$[\text{base } b], \Sigma \stackrel{C}{\to} b, \Sigma \quad (\textsc{CBase})$$

$$\frac{\begin{array}{c} \Sigma = (\Gamma, S) \\ e_1, (\Gamma_1, S) \to v_1, \Sigma_2 \end{array}}{[\text{var } \ \langle \Gamma_1, \_, e_1 \rangle], \Sigma \stackrel{C}{\to} v_1} \quad (\textsc{CVar})$$

$$\frac{a_1, \Sigma \stackrel{C}{\to} e_1 \qquad a_2, \Sigma \stackrel{C}{\to} e_2}{[\text{op}\, a_1\, a_2], \Sigma \stackrel{C}{\to} e_1 \ \text{op} \ e_2} \quad (\textsc{COp})$$

$$\frac{a, \Sigma \stackrel{C}{\to} e}{[\text{fn } x\, a], \Sigma \stackrel{C}{\to} \langle \Gamma, x, e \rangle} \quad (\textsc{CFn})$$

Figure 2: Rules $\to$ for the evaluation of Lua expressions, $\stackrel{D}{\to}$ for decompiling Lua expressions into ASTs, and $\stackrel{C}{\to}$ for compiling ASTs back into expressions.

References

[1] DeVito et al. "Terra: a multi-stage language for high-performance computing" PLDI'13.