CMSC 22610	**Implementation**	**Project 4**
Winter 2004	**of**	**February 23, 2004**
	Computer Languages	

Mini-Lua interpreter

Due: March 12, 2004

1 Introduction

The last programming project is the implementation of an interpreter for Mini-Lua. This part of the project involves two steps. First, the abstract syntax tree (AST) produced in Part 3 must be converted to *executable tree* (ET) format. Then you must implement an interpreter for ET.

2 A core semantics for Mini-Lua

To guide your implementation effort, we describe in this section an operational semantics for a significant subset of Mini-Lua.

2.1 Abstract syntax

The dynamic semantics of Mini-Lua is given in terms of the abstract syntax of a core subset of the language. This syntax is given in Figure 1. We use VAR to denote the set of Mini-Lua variables and STMT to denote the set of terms that represent statements.

2.2 Locations

Mini-Lua is an imperative language, so need a notion of location in our semantics. Environments map variables to locations, while the store maps locations to values.

$$\begin{array}{rcll} \rho & \in & \textsc{Loc} & \text{locations} \\ \Gamma & \in & \textsc{Env} = \textsc{Var} \stackrel{\text{fin}}{\to} \textsc{Loc} & \text{environments} \\ \Sigma & \in & \textsc{Store} = \textsc{Loc} \stackrel{\text{fin}}{\to} \textsc{Value} & \text{stores} \end{array}$$

2.3 Values

Values in Mini-Lua are either primitive (**nil**, booleans, numbers, or strings), functions, or tables. Functions are represented by closures, while tables are represented by locations, which in turn are

$$
\begin{array}{rcl}
s & ::= & s_1\,\texttt{;}\,s_2 \\
  & | & l\ \texttt{=}\ e \\
  & | & e(e_1,\,\ldots,\,e_n) \\
  & | & \texttt{do}\ s\ \texttt{end} \\
  & | & \texttt{while}\ e\ \texttt{do}\ s \\
  & | & \texttt{if}\ e\ \texttt{then}\ s_1\ \texttt{else}\ s_2 \\
  & | & \texttt{return}\ e \\
  & | & \texttt{function}\ f\,\texttt{(}x_1,\,\ldots,\,x_n)\ s \\
  & | & \texttt{local}\ \texttt{function}\ f\,\texttt{(}x_1,\,\ldots,\,x_n)\ s \\
  & | & \texttt{local}\ x\ \texttt{=}\ e \\
 & & \\
l & ::= & x \\
  & | & e\texttt{[}e\texttt{]} \\
 & & \\
e & ::= & b \\
  & | & l \\
  & | & e(e_1,\,\ldots,\,e_n) \\
  & | & \texttt{\{ [}e_1\texttt{]=}e'_1,\,\ldots,\,\texttt{[}e_n\texttt{]=}e'_n\ \texttt{\}} \\
 & & \\
b & ::= & \mathbf{nil} \mid \mathbf{true} \mid \mathbf{false} \mid \cdots
\end{array}
$$

Figure 1: Abstract syntax for Core Mini-Lua

mapped to finite functions from values to locations.

$$
\begin{array}{rcll}
v & \in & \textsc{Value} = \textsc{Prim} \cup \textsc{Clos} \cup \textsc{Loc} \cup \textsc{Table} & \text{values} \\
b & \in & \textsc{Prim} = \{\mathbf{nil},\, \mathbf{true},\, \mathbf{false},\, \ldots\} & \text{primitive values} \\
[\Gamma, (x_1,\,\ldots,\,x_n), s] & \in & \textsc{Clos} = \textsc{Env} \times \textsc{Var}^* \times \textsc{Stmt} & \text{closures} \\
\Theta & \in & \textsc{Table} = (\textsc{Value} \setminus \{\mathbf{nil}\}) \overset{\text{fin}}{\to} \textsc{Loc} & \text{tables}
\end{array}
$$

We use the notation $[\Gamma, (x_1,\,\ldots,\,x_n), s]$ for a closure with environment Γ, parameters $x_1,\,\ldots,\,x_n$, and body s.

2.4 Evaluation judgments

The dynamic semantics of Mini-Lua are specified using five evaluation judgment forms. For programs, the judgment

$$\Sigma, \Gamma \vdash p \Downarrow$$

states that starting with an initial store Σ and initial environment Γ, the program p runs to completion. For statement evaluation, we have a small complication that is required to handle function returns. The result of evaluating an expression is either a store/environment pair

$$\Sigma, \Gamma \vdash s \Rightarrow \Sigma', \Gamma'$$

or a return value/store pair

$$\Sigma, \Gamma \vdash s \Rightarrow \mathbf{Ret}(v, \Sigma')$$

We use R to denote the result of evaluating a statement, when the form it takes does not matter. Function applications for both statements and expressions are described by the application evaluation judgment.

$$\Sigma, \Gamma \vdash e(e_1,\, \ldots,\, e_n) \Rightarrow R$$

Expression evaluation is broken into two evaluation judgment forms. One for evaluating left-hand-side expressions, which return locations,

$$\Sigma, \Gamma \vdash l \Rightarrow \rho, \Sigma'$$

and one for evaluation right-hand-side expression

$$\Sigma, \Gamma \vdash e \Rightarrow v, \Sigma'$$

2.5 Program evaluation

Let $\mathrm{Global}(p)$ be the set of global variables in the program p. Then the rule for program evaluation is:

$$\frac{\Sigma_0 = \{\rho_x \mapsto \mathbf{nil} \mid x \in \mathrm{Global}(p)\} \quad \Gamma_0 = \{x \mapsto \rho_x \mid x \in \mathrm{Global}(p)\} \quad \Sigma_0, \Gamma_0 \vdash p \Rightarrow \Sigma', \Gamma'}{\Sigma_0, \Gamma_0 \vdash p \Downarrow}$$

2.6 Statement evaluation

We evaluate sequences of statements from left to right, propagating the store and environment. If we hit a **`return`** statement, then evaluation is short-circuited.

$$\frac{\Sigma, \Gamma \vdash s_1 \Rightarrow \Sigma_1, \Gamma_1 \quad \Sigma_1, \Gamma_1 \vdash s_2 \Rightarrow \Sigma_2, \Gamma_2}{\Sigma, \Gamma \vdash s_1 \,\texttt{;}\, s_2 \Rightarrow \Sigma_2, \Gamma_2}$$

$$\frac{\Sigma, \Gamma \vdash s_1 \Rightarrow \mathbf{Ret}(v, \Sigma')}{\Sigma, \Gamma \vdash s_1 \,\texttt{;}\, s_2 \Rightarrow \mathbf{Ret}(v, \Sigma')}$$

Assignment modifies the store, but has no effect on the environment.

$$\frac{\Sigma, \Gamma \vdash l \Rightarrow \rho, \Sigma' \quad \Sigma', \Gamma \vdash e \Rightarrow v, \Sigma''}{\Sigma, \Gamma \vdash l \,\texttt{=}\, e \Rightarrow \Sigma'' \pm \{\rho \mapsto v\}, \Gamma}$$

There are two cases for function evaluation, depending on if the function returns a value.

$$\frac{\Sigma, \Gamma \vdash e(e_1,\, \ldots,\, e_n) \Rightarrow \mathbf{Ret}(v, \Sigma')}{\Sigma, \Gamma \vdash e(e_1,\, \ldots,\, e_n) \Rightarrow \Sigma', \Gamma}$$

$$\frac{\Sigma, \Gamma \vdash e(e_1,\, \ldots,\, e_n) \Rightarrow \Sigma', \Gamma'}{\Sigma, \Gamma \vdash e(e_1,\, \ldots,\, e_n) \Rightarrow \Sigma', \Gamma}$$

Blocks limit the scope of the environment.

$$\frac{\Sigma, \Gamma \vdash s \Rightarrow \Sigma', \Gamma'}{\Sigma, \Gamma \vdash \textsf{do}\ s\ \textsf{end} \Rightarrow \Sigma', \Gamma}$$

$$\frac{\Sigma, \Gamma \vdash s \Rightarrow \mathbf{Ret}(v, \Sigma')}{\Sigma, \Gamma \vdash \textsf{do}\ s\ \textsf{end} \Rightarrow \mathbf{Ret}(v, \Sigma')}$$

While loops terminate when either the condition is **false**, or a **`return`** statement is executed in the loop body.

$$\frac{\Sigma, \Gamma \vdash e \Rightarrow \mathbf{false}, \Sigma'}{\Sigma, \Gamma \vdash \texttt{while}\ e\ \texttt{do}\ s \Rightarrow \Sigma', \Gamma}$$

$$\frac{\Sigma, \Gamma \vdash e \Rightarrow \mathbf{true}, \Sigma' \quad \Sigma', \Gamma \vdash s \Rightarrow \Sigma'', \Gamma'' \quad \Sigma'', \Gamma \vdash \texttt{while}\ e\ \texttt{do}\ s \Rightarrow \Sigma''', \Gamma}{\Sigma, \Gamma \vdash \texttt{while}\ e\ \texttt{do}\ s \Rightarrow \Sigma''', \Gamma}$$

$$\frac{\Sigma, \Gamma \vdash e \Rightarrow \mathbf{true}, \Sigma' \quad \Sigma', \Gamma \vdash s \Rightarrow \mathbf{Ret}(v, \Sigma'')}{\Sigma, \Gamma \vdash \texttt{while}\ e\ \texttt{do}\ s \Rightarrow \mathbf{Ret}(v, \Sigma'')}$$

If statements test their condition and then execute the appropriate branch.

$$\frac{\Sigma, \Gamma \vdash e \Rightarrow \mathbf{true}, \Sigma' \quad \Sigma', \Gamma \vdash s_1 \Rightarrow \Sigma_1, \Gamma_1}{\Sigma, \Gamma \vdash \texttt{if}\ e\ \texttt{then}\ s_1\ \texttt{else}\ s_2 \Rightarrow \Sigma_1, \Gamma}$$

$$\frac{\Sigma, \Gamma \vdash e \Rightarrow \mathbf{true}, \Sigma' \quad \Sigma', \Gamma \vdash s_1 \Rightarrow \mathbf{Ret}(v, \Sigma_1)}{\Sigma, \Gamma \vdash \texttt{if}\ e\ \texttt{then}\ s_1\ \texttt{else}\ s_2 \Rightarrow \mathbf{Ret}(v, \Sigma_1)}$$

$$\frac{\Sigma, \Gamma \vdash e \Rightarrow \mathbf{false}, \Sigma' \quad \Sigma', \Gamma \vdash s_2 \Rightarrow \Sigma_2, \Gamma_2}{\Sigma, \Gamma \vdash \texttt{if}\ e\ \texttt{then}\ s_1\ \texttt{else}\ s_2 \Rightarrow \Sigma_2, \Gamma}$$

$$\frac{\Sigma, \Gamma \vdash e \Rightarrow \mathbf{false}, \Sigma' \quad \Sigma', \Gamma \vdash s_2 \Rightarrow \mathbf{Ret}(v, \Sigma_2)}{\Sigma, \Gamma \vdash \texttt{if}\ e\ \texttt{then}\ s_1\ \texttt{else}\ s_2 \Rightarrow \mathbf{Ret}(v, \Sigma_1)}$$

The return statement returns a value/store pair.

$$\frac{\Sigma, \Gamma \vdash e \Rightarrow v, \Sigma'}{\Sigma, \Gamma \vdash \texttt{return}\ e \Rightarrow \mathbf{Ret}(v, \Sigma')}$$

A function definition modifies the store to map the function name to a newly formed closure value.

$$\frac{\rho = \Gamma(f) \quad \Gamma_f = \Gamma \downarrow (\mathrm{FV}(s) \setminus \{x_1, \ldots, x_n\})}{\Sigma, \Gamma \vdash \texttt{function}\ f\,(x_1, \ldots, x_n)\ s \Rightarrow \Sigma \pm \{\rho \mapsto [\Gamma_f, (x_1, \ldots, x_n), s]\}, \Gamma}$$

A local function definition binds the function name to a new location, which is initialized to hold a closure.

$$\frac{\rho \notin \mathrm{dom}(\Sigma) \quad \Gamma_f = \Gamma \downarrow (\mathrm{FV}(s) \setminus \{x_1, \ldots, x_n\})}{\Sigma, \Gamma \vdash \texttt{local function}\ f\,(x_1, \ldots, x_n)\ s \Rightarrow \Sigma \pm \{\rho \mapsto [\Gamma_f, (x_1, \ldots, x_n), s]\}, \Gamma}$$

A local variable declaration binds the variable name to a new location, which is initialized to hold the value of the right-hand side expression.

$$\frac{\Sigma, \Gamma \vdash e \Rightarrow v, \Sigma' \quad \rho \notin \mathrm{dom}(\Sigma')}{\Sigma, \Gamma \vdash \texttt{local}\ x\ \texttt{=}\ e \Rightarrow \Sigma' \pm \{\rho \mapsto v\}, \Gamma \pm \{x \mapsto \rho\}}$$

2.7 Function application evaluation

Applying a function requires first evaluating the function expression and argument expressions from left to right and then applying the function's closure to the argument values.

$$\frac{\begin{array}{c}\Sigma, \Gamma \vdash e \Rightarrow [\Gamma', (x_1, \ldots, x_n), s], \Sigma' \\ \Sigma', \Gamma \vdash e_1 \Rightarrow v_1, \Sigma_1 \quad \cdots \quad \Sigma_{n-1}, \Gamma \vdash e_n \Rightarrow v_n, \Sigma_n \\ \Sigma_n, \Gamma' \pm \{x_1 \mapsto v_1, \ldots, x_n \mapsto v_n\} \vdash s \Rightarrow R\end{array}}{\Sigma, \Gamma \vdash e(e_1, \ldots, e_n) \Rightarrow R}$$

2.8 Left-hand side evaluation

Left-hand side expressions evaluate to locations. For variables, this means looking up the location in the environment.

$$\frac{\rho = \Gamma(x)}{\Sigma, \Gamma \vdash x \Rightarrow \Sigma(\rho), \Sigma}$$

Tables map values to locations; if no mapping exists, a new one is added.

$$\frac{\Sigma, \Gamma \vdash e_1 \Rightarrow \rho, \Sigma_1 \quad \Sigma_1, \Gamma \vdash e_2 \Rightarrow v, \Sigma_2 \quad \Sigma_2(\rho) = \Theta \quad v \in \mathrm{dom}(\Theta)}{\Sigma, \Gamma \vdash e_1 \mathbf{[} e_2 \mathbf{]} \Rightarrow \Theta(v), \Sigma_2}$$

$$\frac{\begin{array}{c}\Sigma, \Gamma \vdash e_1 \Rightarrow \rho, \Sigma_1 \quad \Sigma_1, \Gamma \vdash e_2 \Rightarrow v, \Sigma_2 \\ \Sigma_2(\rho) = \Theta \quad v \notin \mathrm{dom}(\Theta) \quad v \neq \mathbf{nil} \quad \rho' \notin \mathrm{dom}(\Sigma_2)\end{array}}{\Sigma, \Gamma \vdash e_1 \mathbf{[} e_2 \mathbf{]} \Rightarrow \rho', \Sigma_2 \pm \{\rho \mapsto \Theta \pm \{v \mapsto \rho'\},\ \rho' \mapsto \mathbf{nil}\}}$$

2.9 Expression evaluation

Primitive values evaluate to themselves.

$$\frac{}{\Sigma, \Gamma \vdash b \Rightarrow b, \Sigma}$$

Lef-hand-side values evaluate to the value stored at their location.

$$\frac{\Sigma, \Gamma \vdash l \Rightarrow \rho, \Sigma'}{\Sigma, \Gamma \vdash l \Rightarrow \Sigma'(\rho), \Sigma'}$$

A function call evaluates to its return value (if there is no return value, then a runtime error has occurred).

$$\frac{\Sigma, \Gamma \vdash e(e_1,\ \ldots,\ e_n) \Rightarrow \mathbf{Ret}(v, \Sigma')}{\Sigma, \Gamma \vdash e(e_1,\ \ldots,\ e_n) \Rightarrow v, \Sigma'}$$

Table expressions are evaluated from left to right (note that if multiple fields have the same index, then the rightmost field will define the value). The result of a table expression is the table's location.

$$\frac{\begin{array}{c}\Sigma, \Gamma \vdash e_1 \Rightarrow v_1, \Sigma_1 \quad \Sigma_1, \Gamma \vdash e'_1 \Rightarrow v'_1, \Sigma'_1 \\ \cdots \\ \Sigma'_{n-1}, \Gamma \vdash e_n \Rightarrow v_n, \Sigma_n \quad \Sigma_n, \Gamma \vdash e'_n \Rightarrow v'_n, \Sigma'_n \\ v_i \neq \mathbf{nil} \text{ for } i \in [1..n] \\ \rho, \rho_1,\ \ldots,\ \rho_n \notin \mathrm{dom}(\Sigma'_n) \text{ are unique} \\ \Theta = \{v_1 \mapsto \rho_1\} \pm \cdots \pm \{v_n \mapsto \rho_n\} \\ \Sigma' = \Sigma'_n \pm \{\rho \mapsto \Theta,\ \rho_1 \mapsto v'_1,\ \ldots,\ \rho_n \mapsto v'_n\}\end{array}}{\Sigma, \Gamma \vdash \mathbf{\{\ [} e_1 \mathbf{]=} e'_1,\ \ldots,\ \mathbf{[} e_n \mathbf{]=} e'_n \mathbf{\ \}} \Rightarrow \rho, \Sigma'}$$

3 Builtin functions

Your implementation should include support for the following builtin Mini-Lua functions:

`error` *(msg)*

Print the message *msg* to the standard error stream and terminate the Mini-Lua interpreter. This function does not return any results.

`loadfile` *(filename)*

Loads a file as a Mini-Lua chunk (without running it). If no errors are returned, then it returns the compiled chunk as a function, otherwise it returns **nil**. Errors loading the file are reported to the standard error stream. Note that this operation will extend the global environment of the program.

`next` *(table, index)*

Return the next element of *table* after *index*. The order of table entries is undefined. If **nil** is given as the index, then the first element in the table is returned.

`print` *(s)*

prints the string *s* to the standard output. This function does not return any results.

`tonumber` *(s)*

converts *s*, which should be a decimal string, to a number. Leading and trailing whitespace is ignored; **nil** is returned if there is an error.

`tostring` *(value)*

Returns a string representation of *value*. For primitive values, this string should be the representation of the value, for functions it should be `"<function>"` and for tables it should be `"<table>"`.

`type` *(value)*

Returns the type of *value* encoded as a string. The possible results are: `"nil"`, `"number"`, `"string"`, `"boolean"`, `"table"`, and `"function"`.

`string.byte` *(s,i)*

Returns the integer value of the *i*th character in the string *s*. Returns **nil** if there is an error.

`string.len` *(s)*

Returns the length of the string *s* or else **nil** if *s* is not a string.

`string.sub` *(s,i,j)*

Returns the substring of *s* from index *i* to *j* (inclusive). If *i* is greater than *j*, then it returns the empty string. It returns **nil** if there is a type error.

4 Requirements

We will supply the execution tree representation. Your job will be to do the following three things:

- Extend your typechecker to handle the builtin functions.
- Translate the AST produced by your typechecker to the execution tree format.
- Write an interpreter for the execution tree format.

The assignment is due on the last day of classes (March 12).