CMSC 22610	**Implementation**	**Handout 2**
Winter 2004	**of**	**January 9, 2004**
	Computer Languages	

Project overview

1 Introduction

The project for this course is the implementation of an interpreter for *Mini-Lua*, which is a subset of the scripting language Lua.[1] The project will be broken into four pieces (or milestones). Roughly half of the project grade will depend on the milestones and half on the completed project. This document gives an overview of the project; separate handouts will describe the individual pieces of the project.

2 Mini Lua

Lua is dynamically typed, higher-order, scripting language with a Pascal-like syntax. It supports basic values, such as booleans, numbers, and strings, as well as associative arrays (called *tables*). Tables are used to implement data structures and provide support for an object-oriented style of programming.

We give an overview of the language features in the remainder of this section. More precise descriptions of the syntax and semantics of Mini-Lua will be given in subsequent handouts.

2.1 Mini-lua values

Mini-Lua is a dynamically typed language, which means that variables do not have types; only values do. There are no type definitions in the language. All values carry their own type and type errors are signaled at runtime.

There are six basic types in Mini-Lua: nil, boolean, integer, string, function, and table. The first four of these are the types of atomic values in Mini-Lua. Nil is the type of the value **`nil`**, whose main property is to be different from any other value; usually it represents the absence of a useful value. Boolean is the type of the values **`false`** and **`true`**. In Mini-Lua, both **`nil`** and **`false`** make a condition false; any other value makes it true. Integer is the type of arbitrary precision integers. Number represents real (double-precision floating-point) numbers. String represents arrays of 8-bit characters. Strings may contain any 8-bit character, including embedded zeros. Functions are first-class values in Mini-Lua, which means that they can be stored in variables, passed as arguments to other functions, and returned as results.

[1] The Lua manual is available online at `www.lua.org`.

The type table represents associative arrays, that is, arrays that can be indexed not only with numbers, but with any atomic value (except **`nil`**). Moreover, tables can be heterogeneous, that is, they can contain any non-nil value. Tables are the sole data structuring mechanism in Lua; they may be used to represent ordinary arrays, symbol tables, sets, records, graphs, trees, etc. To represent records, Lua uses the field name as an index. The language supports this representation with syntactic sugar.

Like indices, the value of a table field can be of any atomic type (except **`nil`**). In particular, because functions are first class values, table fields may contain functions. In this case, they are called *methods*. Mini-Lua provides syntactic sugar for using tables to program in an object-oriented style.

Tables and functions are *objects*: variables do not actually contain these values, only references to them. Assignment, parameter passing, and function returns always manipulate references to such values; these operations do not imply any kind of copy.

2.2 Variables

Variables are mutable locations that store values (what are often called *l-values*). There are three kinds of variables in Mini-Lua: global variables, which have scope over the entire program; local variables, which have scope local to the block they are defined in, and table fields. Local and global variables are denoted by a single name. The field of a table is denoted using array-indexing notation (*e.g.*, *a*`[`*i*`]`). Mini-Lua uses string-valued indices to model labeled fields and provides the notation

$$Exp\texttt{.}\mathrm{Name}$$

as shorthand for

$$Exp\texttt{["}\mathrm{Name}\texttt{"]}$$

2.3 Statements

Mini-Lua supports most of the statement forms found in Lua. These include assignment, control structures, procedure calls, table constructors, and variable declarations (see Figure 1).

2.4 Expressions

Mini-Lua supports the full range expression forms found in Lua, with the exception of a more limited syntax for table fields (see Figure 2).

2.5 Functions

Functions in Mini-Lua are first-class, which means that they can be passed as arguments, returned as function results, and stored in tables. Named function declarations are a statement form (see Figure 1), but Mini-Lua also provides anonymous functions as an expression form. The syntax of a function is

Function
::= **`function`** *FunctionBody*

Block
::= (*Stmt* **;**)*

Stmt
::= *Vars* **=** *Exps*
| *FunctionCall*
| **do** *Block* **end**
| **while** *Exp* **do** *Block* **end**
| **if** *Exp* **then** *Block* (**elseif** *Exp* **then** *Block*)* (**else** *Block*)^opt **end**
| **return** *Exps*^opt
| **break**
| **for** Name **=** *Exp* **,** *Exp* (**,** *Exp*)^opt **do** *Block* **end**
| **for** *Names* **in** *Exps* **do** *Block end*
| **local**^opt **function** Name *FunctionBody*
| **local** *Names* **=** *Exps*

Figure 1: The syntax of Mini-Lua statements

FunctionBody
::= **(** *Params*^opt **)** *Block* **end**

Like SML and Scheme, Lua is a lexically-scoped language, which means that function values are represented by closures.

Function calls are both a statement and expression form. The syntax of a function call is:

FunctionCall
::= *PrefixExp Args*
| *PrefixExp* **:** Name *Args*

Args
::= **(** *Exps*^opt **)**
| **{** (*Field* (**,** *Field*)*)^opt **}**

The second form for functions arguments is syntactic sugar for applying a function to a single table argument.

The fact that functions can be stored in tables allows them to be used as *methods*. The syntax

PrefixExp **:** Name **(** ... **)**

is used for method dispatch. It evaluates as

let *obj* = *PrefixExp* **in** *obj*.Name(*obj*, ...)

2.6 Differences with Lua

The syntax and semantics of Mini-Lua are a subset of full Lua. Here is a list of the major differences:

Exps
::= *Exp* (**,** *Exp*)*

Exp
::= *Exp BinOp Exp*
| **not** *Exp*
| **-** *Exp*
| *PrefixExp*
| *Function*
| **{** (*Field* (**,** *Field*)*)opt **}**
| **nil**
| **true**
| **false**
| Number
| String

PrefixExp
::= *Var*
| *FunctionCall*
| **(** *Exp* **)**

Field
::= **[** *Exp* **]** **=** *Exp*
| Name **=** *Exp*

BinOp
::= **or** | **and** | **<** | **>** | **<=** | **>=** | **~=** | **==** | **..** | **+** | **-** | ***** | **/** | **^**

Figure 2: The syntax of Mini-Lua expressions

- Mini-Lua does not allow runtime coercions between strings and integers.
- Mini-Lua does not have *mata-tables*.
- Numbers in Mini-Lua are integers, not reals.
- There are a number of syntatic forms in Lua that are not in Mini-Lua: repeat-loops, field-names for function definitions, implicitly index field definitions, and other quote characters for string literals.

3 Project schedule

The following is a tentative schedule for the project assignments.

Assignment date	Description	Due date
Jan. 7	Lexer	Jan. 23
Jan. 21	Parser	Feb. 4
Jan. 28	Analyser	Feb. 13
Feb. 16	Code generation and runtime	Mar. 12