**CMSC 22610** **Implementation** **Project 3**
**Winter 2004** **of** **February 4, 2004**
**Computer Languages**

# Mini-Lua analyser

**Due: February 16, 2004**

## 1 Introduction

Your third assignment is to implement an sytactic analyser for Mini-Lua. Since Mini-Lua is a dynamically typed language, this analyser is not responsible for typechecking. The main job of the analyser is to identify the binding sites of variables and to determine the free variables of function definitions. It also needs to check some additional syntactic correctness properties that are not caught by the parser. The result of analysis will be a *typed abstract syntax tree* (or AST for short) and associated symbol table. The AST and symbol table should have the following properties:

- Each distinct named variable should have a symbol table entry, which marks the variable as local or global.
- Each variable occurrence should refer to the corresponding table entry.
- Each function definition should be annotated with the free variables in the function.
- Derived forms (see below) should be represented by their expansion.

## 2 Correctness properties

In addition to resolving the bindings of identifiers, your program should check for the following possible errors:

- undeclared variables (all variables should be declared before use)
- **`break`** and **`continue`** statements outside loops
- table expressions that define the same field multiple times
- multiple occurrences of a name in a function parameter list

Your program should report the location of the error and continue checking the input.

## 3 Derived forms

In the abstract syntax, we have expanded various derived forms from the concrete syntax into their semantic equivalents. This expansion simplifies the intermediate representation and reduces the work of code generation.

We use the following expansions:

- The variable form "$e$`.`$f$" is replaced with "$e$`["`$f$`"]`".
- The field form "$f$ `=` $e$" is replaced with "`["`$f$`"] =` $e$".
- The function argument form "`{` *Fields* `}`" is replaced with "`({` *Fields* `})`".
- The statement form

  **`for`** $i$ **`=`** $e_1$`,` $e_2$ **`do`** *Block* **`end`**

  is replaced with

  **`for`** $i$ **`=`** $e_1$`,` $e_2$`, 1` **`do`** *Block* **`end`**

- The statement form

  **`for`** $i$ **`=`** $e_1$`,` $e_2$`,` $e_3$ **`do`** *Block* **`end`**

  is replaced with

```
do local
  start, stop, step = e1, e2, e3
  function iter (x, y) {
    if (x == stop) return nil;
    x = x + step;
    return x
  }
  for i in (iter, start, nil) do Block end
end
```

  Note that here we are assuming that `start`, `stop`, `step`, and `iter` are fresh variables.

## 4 Requirements

Your implementation should include a file `main.sml` that defines the structure

```
structure Main : sig
  val main : (string * string list) -> OS.Process.status
end = ...
```

The second argument to `main` will be the command-line arguments; your program should treat them as file names. Assuming that your CM file is called `mini-lua.cm`, you can compile your program using the following shell command:

```
ml-build mini-lua.cm Main.main mini-lua
```

This command will produce a heap file that you can run using the command

```
sml @SMLload=mini-lua
```