# Lua: Advanced Flow
# Overview of Lexing & Parsing

CS F331 Programming Languages
CSCE A331 Programming Language Concepts
Lecture Slides
Monday, February 4, 2019

Glenn G. Chappell
Department of Computer Science
University of Alaska Fairbanks
ggchappell@alaska.edu

© 2017–2019 Glenn G. Chappell

## Review
## Introduction to Lua

The **Lua** PL originated in 1993 in Brazil.

Lua’s **source tree** is small, easy to include in other projects. It is a popular scripting language for games, LaTeX, Wikipedia, etc.

Characteristics

- Dynamic PL.
- Simple syntax. Very little **punctuation**. Small, versatile feature set.
- **Imperative**.
- Insulated from machine.
- Typing: **dynamic**, **implicit**. **Duck typing**.
- Eight types: `number`, `string`, `boolean`, `table`, `function`, `nil`, `userdata`, `thread`.
- **First-class functions**.
- Function definitions are executable statements.
- Uses **eager** evaluation (opposite: **lazy** evaluation).

## Review
## Lua: Organization — Modules [1/2]

Lua **module**: an importable file—the kind of thing we would make a header/source combination for in C++.

To import a module into a program, use function `require`:

```
mymod = require "mymod"
```

Since we pass only a string literal, we may leave off the parentheses. This is common.

Giving the return value the same name as the module makes its purpose clear.

Then access module members with the dot operator:

```
io.write(mymod.add6(15).."\n");
```

The module code is in a file whose name is the module name plus the Lua suffix: "`mymod.lua`".

*See* `mymod.lua` & `org.lua`.

# Lua: Organization — Modules [2/2]

A module is like a function returning a table containing module members. In the module source, initialize the table as empty:

```
local mymod = {}
```

Things to **export** are table members. Everything else is local.

```
function mymod.add6(n)
  return n+6
end
```

Return the module table at the end of the file.

```
return mymod
```

*See* `mymod.lua` & `org.lua`.

# Lua: Organization — Metatables

A Lua table can have an associated **metatable**.

```
setmetatable(t, mt)  -- Make mt the metatable for t
```

An attempt to access a nonexistent key in a table causes function `__index` in the metatable to be called. Its return value is returned as the associated value for the given key.

```
function mt.__index(tbl, key)
  return mt[key]
end
```

This can be used to implement something like classes & objects.

*See* `mymod.lua` & `org.lua`.

## Review
## Lua: Organization — Colon Operator

When a Lua function lies in a table, the function does not know this—or what table it lies in. In order to allow the function to act on the table (as if it were a member function of an object), we can pass the table as a parameter.

The colon operator offers a convenient way to do this.

```
tbl:key(x, y)
tbl.key(tbl, x, y)
```

The above do the same thing.

*See* `mymod.lua` & `org.lua`.

# Lua: Organization — Closures

A **closure** is a function that carries with it (some portion of) the environment in which it was defined.

A closure can form a simpler alternative to traditional OO constructions (classes, objects), particularly when a class exists primarily to support a single member function.

Closures are found in a number of PLs. Since the 2011 standard, C++ has had closures, in the form of *lambda functions*.

*See* `closure.cpp`.

Lua functions are closures. When a function is created, it carries with it the variables available to it at its creation.

If we create a function inside a function or module, and we return the new function, then the local variables of the outer function/module can play the role of private data members.

*See* `org.lua`.

# Lua: Advanced Flow
# Coroutines [1/6]

A **coroutine** is a function that can give up control (we say “**yield**”) at any point, and then later be resumed. Typically, each time a coroutine temporarily gives up control, it passes a value back to its caller (we say it **yields** the value).

A number of PLs feature coroutines prominently.

- Go has the cutesily named *goroutines*.
- Python has long had simple coroutines called *generators*. The yielded values are available to the caller via an iterator. Recently, more general coroutines have been added to Python.
- C++ does not currently have coroutines *per se*, although they can be built using the Standard Library threads facility. It is expected that coroutines will be in the 2020 C++ Standard.

Coroutines are available in Lua through the standard-library module `coroutine`, which is loaded automatically.

# Coroutines [2/6]

The `coroutine` module contains four functions of interest to us.

`yield`

A coroutine *yields*, sending zero or more values to its caller, by calling `coroutine.yield`, passing the value(s) to be yielded, if any. It only `returns` when completely finished; after that, it cannot be resumed.

`create`

Do not call a coroutine directly. Rather, pass the function to `coroutine.create`; the return value is a **coroutine object**, with type `thread`. A coroutine is dealt with only through this coroutine object.

`resume`

To run a coroutine, either starting it or resuming it after a yield, pass the coroutine object to `coroutine.resume`. This returns an error flag (discussed shortly) and the yielded value(s), if any.

`status`

Pass the coroutine object to `coroutine.status` to get its state. This returns a `string`, which is `"dead"` if the coroutine has terminated.

# Lua: Advanced Flow
## Coroutines [3/6]

To *write* a coroutine, write an ordinary Lua function.

Each time you wish to give control back to the caller while allowing for a resume, call `coroutine.yield`, passing the yielded value(s), if any.

If the coroutine is finished, then `return` as usual. Do not return any values. Simply falling off the end of the function will accomplish this.

```
function cfunc()  -- Coroutine; do not call directly
  …
  coroutine.yield(val)
  …
end
```

To *use* a coroutine, first get a coroutine object, by calling `coroutine.create`, passing the coroutine function.

`c` is the coroutine object.

```
c = coroutine.create(cfunc)
```

To execute the coroutine, call `coroutine.resume`, passing the coroutine object. In the first `resume` call, additional arguments are passed to the coroutine function. `coroutine.resume` returns a `boolean` (`false`: error) and the yielded value(s), if any.

```
ok, val = coroutine.resume(c)
```

Now, if the string returned by `coroutine.status` is not `"dead"`, then any yielded value(s) (`val` above) may be used. If further values are desired, call `coroutine.resume` again, as above.

# Lua: Advanced Flow
# Coroutines [5/6]

Q. `coroutine.yield` sends data out of a coroutine. Can we similarly send data *into* a coroutine?

A. Yes! Any additional arguments passed to the second and later calls to `coroutine.resume` become the return value(s) of `coroutine.yield`, inside the coroutine.

Q. Can a coroutine be resumed when its status is `"dead"`?

A. No. To run the coroutine again, we must get a new coroutine object, by calling `coroutine.create` once more.

Q. Must we go through the rigmarole of checking both the error flag (`ok`) and the string returned by `coroutine.status`?

A. If there is an error (`ok` is `false`) then `coroutine.status` will return `"dead"`. So only `coroutine.status` needs to be checked to determine whether to exit a loop going through the yielded values. We *may* wish to check `ok` after leaving the loop.

# Lua: Advanced Flow
# Coroutines [6/6]

Here is a coroutine that yields increasing consecutive integers.

```
-- count1: coroutine.
-- Given a, b. Counts
-- from a up to b.
function count1(a, b)
  while a <= b do
    coroutine.yield(a)
    a = a+1
  end
end
```

Code that uses this coroutine:

```
c = coroutine.create(count1)
ok, val = coroutine.resume(c, 3, 8)
while coroutine.status(c) ~= "dead" do
  io.write(val.." ")
  ok, val = coroutine.resume(c)
end
io.write("\n")

if not ok then  -- OPTIONAL CHECK
  io.write("ERROR in coroutine\n")
end
```

The code at right prints

```
3 4 5 6 7 8
```

# Lua: Advanced Flow
# Custom Iterators [1/5]

Recall: Lua has an iterator-based loop, the `for-in` construction.

```
for k, v in pairs(t) do  -- t is a table
  io.write("Key: "..k..", value: "..v.."\n")
end
```

`pairs` takes a table and returns an iterator, which `for-in` uses.

We can write our own iterators. To do this, we need to understand what code like the following does in Lua.

```
for u, v1, v2 in XYZ do
  FOR_LOOP_BODY
end
```

# Lua: Advanced Flow
# Custom Iterators [2/5]

```
for u, v1, v2 in XYZ do
  FOR_LOOP_BODY
end
```

"`v1, v2`" may be replaced with an arbitrary number of variables, just one variable, or no variables at all.

The above is translated to this:

```
local iter, state, u = XYZ
local v1, v2
while true do
  u, v1, v2 = iter(state, u)
  if u == nil then
    break
  end
  FOR_LOOP_BODY
end
```

`state` is there so that function `iter` can use it to store any data that it wants. But if we take advantage of the fact that `iter` is a closure, then we do not need `state`. We still pass it around, but it can be `nil`.

`u`—and `v1`, `v2`, etc., if they exist—are the values of interest. Function `iter` must return them. The same `u` will be passed to `iter` as its second parameter, but `iter` may ignore it.

# Lua: Advanced Flow
# Custom Iterators [3/5]

Here is a simplified model for creating a Lua iterator.

- The thing we pass to the for-in construction ("`XYZ`" in my example) returns a function ("`iter`") and two other values, which can be `nil`.
- The returned function ("`iter`") must take two parameters, but it may ignore them.
- Function `iter` should return however many values we want at each iteration of the loop.
- Any data that need to be stored between calls to `iter` can be held in variables local to the function that created `iter`. Since `iter` is a closure, it has access to these.
- Signal exhaustion of the iterator—and thus the the end of the for-in loop—by having `iter` do "`return nil`".

# Lua: Advanced Flow
# Custom Iterators [4/5]

When we use this simplified model, a custom iterator might have the following form.

```
function XYZ(…)
  local …
  local function iter(dummy1, dummy2)
    if … then
      return nil  -- Iterator exhausted
    end
    …
    return …      -- Next value(s)
  end
  return iter, nil, nil
end
```

*If needed*, create variables here to store info between calls to `iter`.

# Lua: Advanced Flow
# Custom Iterators [5/5]

Here is an actual custom iterator, based on the simplified model.

```
-- count2: iterator. Given a, b. Counts from a up to b.
function count2(a, b)
  local function iter(dummy1, dummy2)
    if a > b then
      return nil
    end
    local save_a = a
    a = a+1
    return save_a
  end
  return iter, nil, nil
end
```

Code that uses this iterator:

```
for i in count2(3, 8) do
  io.write(i.." ")
end
io.write("\n")
```

The above prints `3 4 5 6 7 8`

## Overview of Lexing & Parsing [1/5]

Here are some of the things a compiler need to do.

1. **Determine whether the given program is syntactically correct, and, if so, find its structure.**
2. Determine all identifiers and what they refer to.
3. Determine types and check that no typing rules are broken—if compiling code in a statically typed PL.
4. Generate code in the target language.

A course on compilers would cover all of the above. Here, we look at item #1, which is called **parsing**.

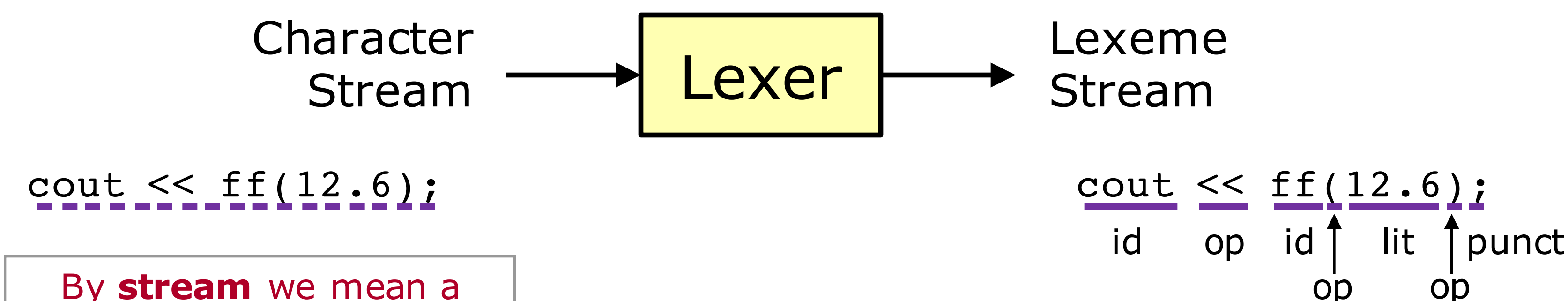
Parsing is often broken into two phases.

- **Lexical analysis**
- **Syntax analysis**

# Overview of Lexing & Parsing [2/5]

**Lexical analysis**, or **lexing**, means breaking up the input into words, which are called **lexemes** (or **tokens**). Lexing takes a stream of characters as input and outputs a stream of lexemes, each usually identified as belonging to a particular **category**.

A code module that does lexical analysis is a **lexical analyzer**, or **lexer**. This generally involves computation at the level of regular grammars and finite automata.



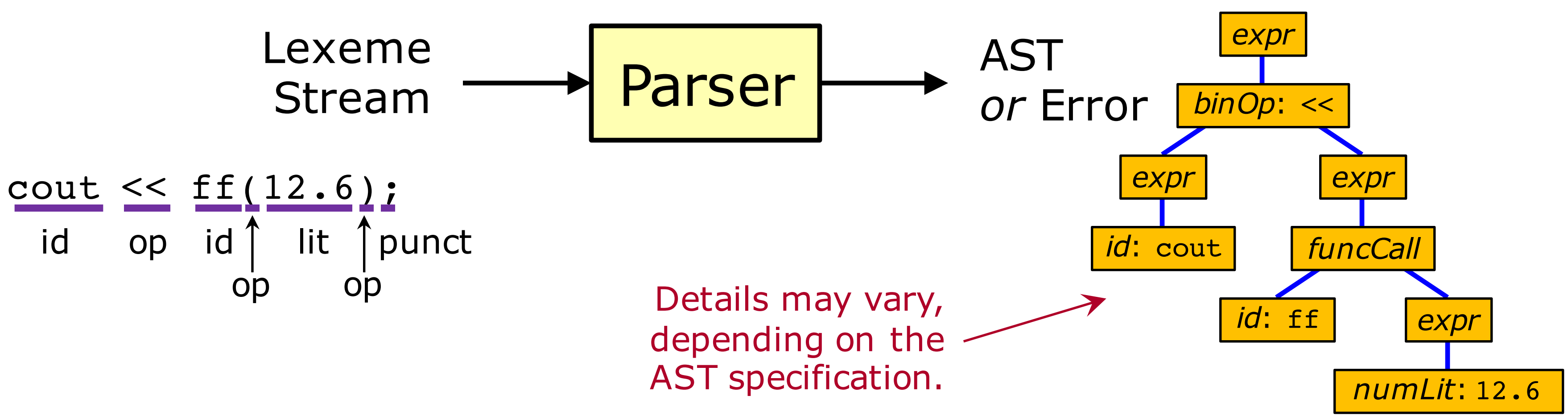

By **stream** we mean a sequence that *might* not ever be stored all at once.

# Overview of Lexing & Parsing [3/5]

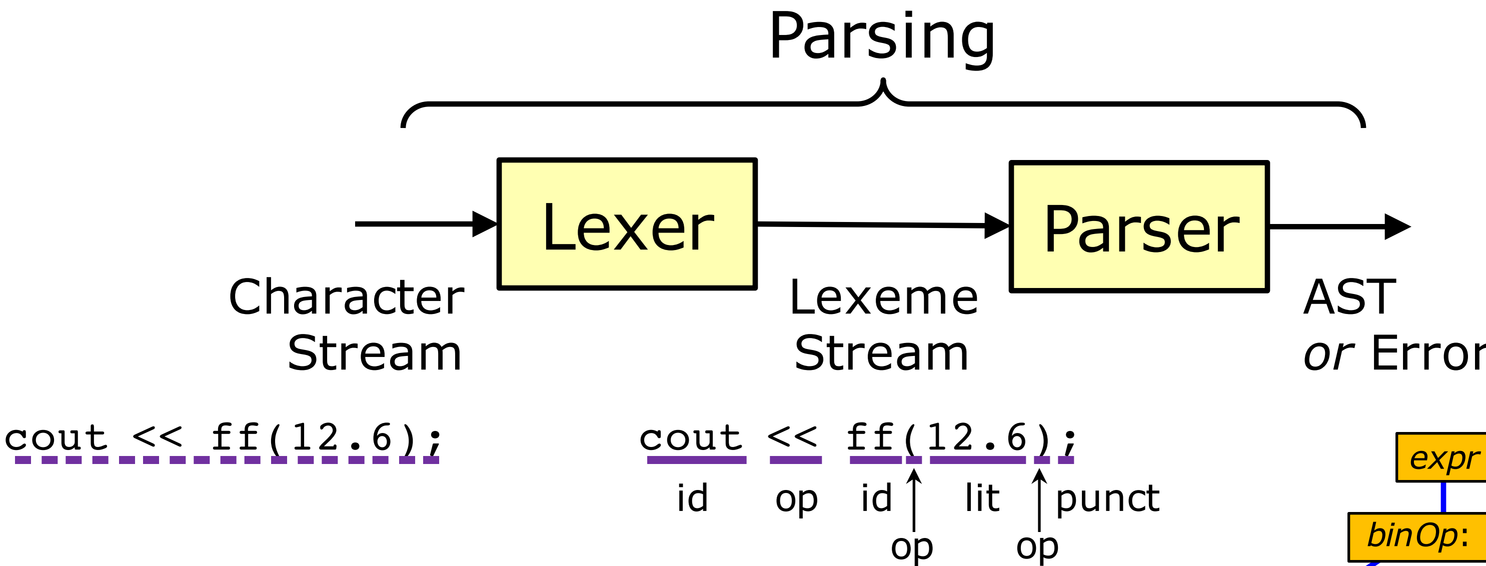
**Syntax analysis**, or **parsing**, takes a stream of lexemes as input, and, if this is syntactically correct, outputs a representation of its structure.

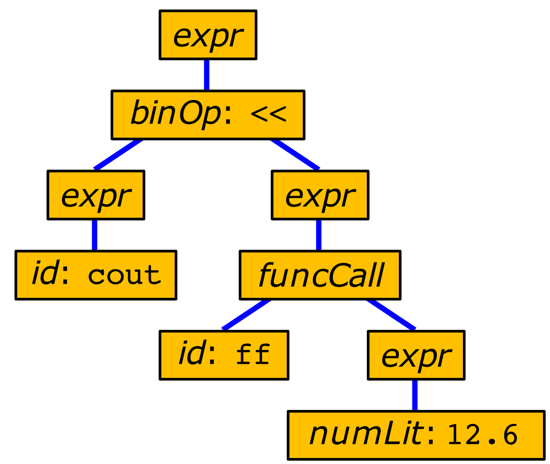
A **parser** involves a higher level of computation than a lexer, at the level of CFGs; there is virtually always a stack used.

The representation that a parser outputs might be a parse tree (concrete syntax tree). But a more common choice is an **abstract syntax tree** (**AST**). Such a tree leaves out things like punctuation, which only serve to guide the parser.



# Overview of Lexing & Parsing [4/5]




Note that we have used the term "parsing" to mean two different things: the entire process, or just the second phase (syntax analysis). So "parsing" has both a general meaning and a specific meaning. In practice, this rarely leads to misunderstandings.

# Overview of Lexing & Parsing [5/5]

Parsing is not always separated into two phases. But such a separation has a number of advantages.

- It makes the code more modular.
- It simplifies lexical analysis, since the more complicated parsing algorithms do not need to be involved in that phase.
- It makes a parser easier to write and more portable, since this code is insulated from the outside world: character sets, files, checking for I/O errors on input, etc.
- It simplifies the implementation of parser **lookahead**: checking one or more lexemes ahead of the current lexeme.