Lua: Advanced Flow
Thoughts on Assignment 2
Introduction to Lexing & Parsing

CS 331 Programming Languages

Lecture Slides

Monday, February 3, 2025

Glenn G. Chappell

Department of Computer Science

University of Alaska Fairbanks

ggchappell@alaska.edu

Unit Overview
The Lua Programming Language

Topics

- ✓ Introduction to survey of programming languages
- ✓ PL feature: compilation & interpretation
- ✓ PL category: dynamic PLs
- ✓ Introduction to Lua
- ✓ Lua: fundamentals
- ✓ Lua: modules
- ✓ Lua: objects
- Lua: advanced flow

Review

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. Little **punctuation**. Small, versatile feature set.
- **Imperative**.
- Typing: **dynamic**, **implicit**. **Duck typing**.
- Fixed set of eight types.
- **First-class functions**.
- Function definitions are executable statements.
- Does **eager evaluation** (alternative: **lazy evaluation**).

Review
Lua: Objects — Metatables, Class & Object

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

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

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

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

For code from this topic, see `pets.lua` *&* `obj.lua`*.*

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

Review
Lua: Objects — 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.

```
tbl.func(tbl, x, y)
```

Lua’s colon operator offers a convenient way to do this.

```
tbl:func(x, y)
```

These two do the same thing.

The colon operator is **syntactic sugar**: syntax that does not add new capabilities, but makes things nicer.

Review
Lua: Objects — Closures

A **closure** is a function that carries with it a reference to or copy of (part of) the environment in which it was created.

A closure can form a simpler alternative to traditional OO constructions (classes, objects), particularly when an object exists primarily to support a single method (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 `obj.lua`*.*

Lua: Advanced Flow

Lua: Advanced Flow
Coroutines — Background [1/3]

A **coroutine** is a function that can temporarily give up control (**yield**) at any point, and then later be **resumed**. Each time a coroutine temporarily gives up control, it may pass one or more values back to its caller (it **yields** these values).

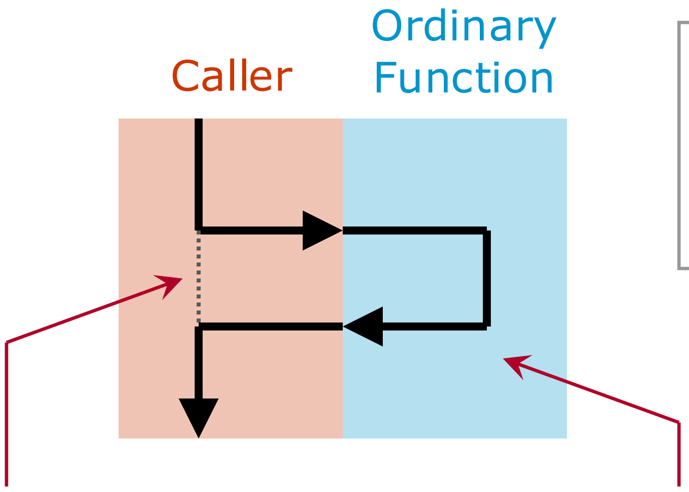
A number of major PLs feature coroutines.

- 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. In 2015, more general coroutines were added to Python.
- Coroutines were added to C++ in the 2020 Standard.
- And—you guessed it—Lua has coroutines. These are available through its standard-library `coroutine` module.

For code from this topic, see `adv.lua`*.*

Lua: Advanced Flow
Coroutines — Background [2/3]

Below is an illustration of the flow of control when an ordinary function is called.



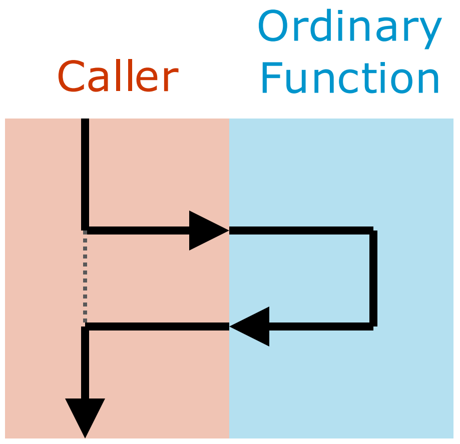
State. Here, this refers to values of local variables and the location in the code where execution is happening.

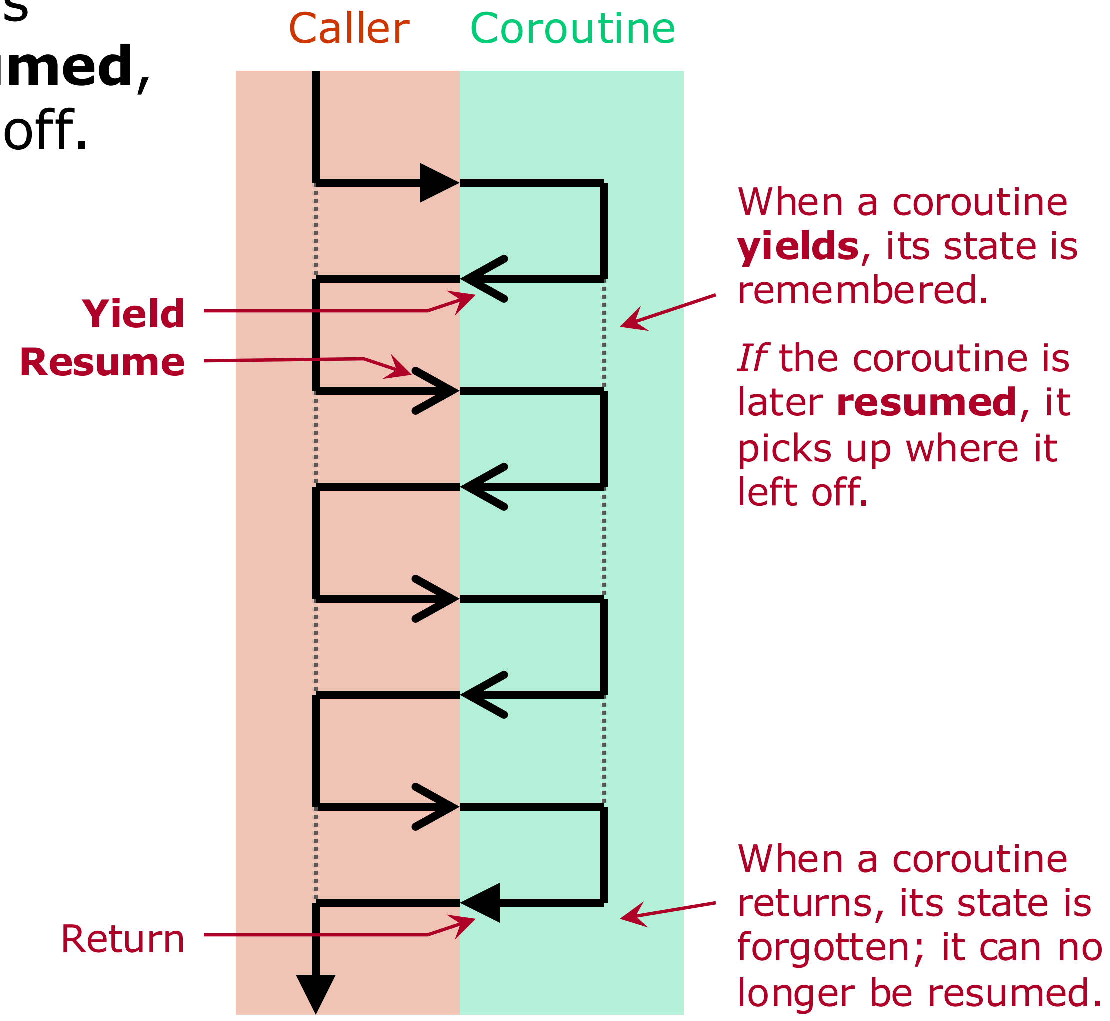
When a function is called, the state of the caller is remembered. After the function returns, the caller picks up where it left off.

When a function returns, the state of the function is forgotten.

Lua: Advanced Flow
Coroutines — Background [3/3]

A coroutine can **yield** to its caller and later be **resumed**, picking up where it left off.





When a coroutine **yields**, its state is remembered.

If the coroutine is later **resumed**, it picks up where it left off.

When a coroutine returns, its state is forgotten; it can no longer be resumed.

Lua: Advanced Flow
Coroutines — In Lua [1/4]

Lua's standard-library `coroutine` module contains two functions of interest to us.

`coroutine.yield`

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

`coroutine.wrap`

In Lua, the caller of a coroutine does not call it directly. Instead, it uses `coroutine.wrap`. Pass a coroutine function to `coroutine.wrap`. The return value is a **coroutine wrapper function**; call this to call/resume the coroutine. The *first* time the wrapper function is called, its arguments—if any—are passed as arguments to the coroutine function.

Lua: Advanced Flow
Coroutines — In Lua [2/4]

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

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

Lua: Advanced Flow
Coroutines — In Lua [3/4]

To *use* a coroutine, first get a **coroutine wrapper function**, by calling `coroutine.wrap`, passing the coroutine function.

`cw` is the coroutine wrapper function.

```
cw = coroutine.wrap(cfunc)
```

To execute the coroutine, call the wrapper function (`cw`, above). The first time you make such a call, pass any arguments you want passed to the coroutine function.

Look at the return value of the wrapper function. If this is `nil` (for multiple return values, if the *first* return value is `nil`), then the coroutine has returned. Otherwise, the return values are the values yielded by the coroutine. Do whatever is appropriate with them. If desired, call the wrapper function again.

Do not call a Lua coroutine directly! Use the `coroutine` module.

Lua: Advanced Flow
Coroutines — In Lua [4/4]

Here is a coroutine that yields increasing consecutive integers.

```
-- count1: coroutine. Given a, b. Counts from a up to b.
function count1(a, b)
    for i = a, b do
        coroutine.yield(i)
    end
end
```

Code that uses this coroutine:

```
cw = coroutine.wrap(count1)
val = cw(3, 8)
while val ~= nil do
    io.write(val.." ")
    val = cw()
io.write("\n")
```

The above prints `3 4 5 6 7 8`

Lua: Advanced Flow
Coroutines — CODE

Suppose we wish to write a coroutine that yields some sequence of values. Here is a method that some people find helpful.

1. Write an ordinary function that *prints* the sequence of values, one by one.
2. Change `print` to `coroutine.yield`.

Similar methods work for writing coroutines in other PLs.

TO DO

- Write a coroutine and code that uses it.

Done. See `adv.lua`*.*

Lua: Advanced Flow
Coroutines — More Advanced

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

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

A coroutine interface along these lines is found in other PLs, too.

Q. Is a coroutine required to yield a value each time it yields?

A. If you are using `coroutine.wrap`, then yes, this is required.

However, Lua has a more general-purpose (and more complicated) interface to coroutines that does not require values to be yielded. Instead of `coroutine.wrap`, this interface uses functions `coroutine.create`, `coroutine.resume`, and `coroutine.status`. See the Lua doc’s.

Lua: Advanced Flow
Custom Iterators [1/6]

In many programming languages, there is a loop construction that does iteration over the values in a container. This might be called a **for-each loop** or an **iterator-based for-loop**.

In Lua, this role is played by the for-in construction.

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

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

We can write our own iterators.

Lua: Advanced Flow
Custom Iterators [2/6]

A Lua **iterator** is a function that is called repeatedly, with each call returning one or more values that are (typically) used in one iteration of a loop. To indicate that the iterator is **exhausted**—that the loop needs to end—the function returns `nil`.

When we do

```
for v in EXPR
    …
end
```

EXPR must be an expression whose value is an iterator—again, a *function*. This function is called repeatedly. If it returns `nil`, then the loop terminates. If it returns something else, then `v` is set to this value, and the loop body is executed.

Lua: Advanced Flow
Custom Iterators [3/6]

A custom iterator might have the following form.

```
function MYFUNCTION(…)
    local …
    local function iter_func()
        if … then
            return nil  -- Iterator exhausted
        end
        …
        return …        -- Return next value(s)
    end
    return iter_func
end
```

`iter_func` is a closure. So we can create variables here to store info between calls to `iter_func`, if we need to.

To use this, do something like

```
for k in MYFUNCTION(…)
    …
end
```

Lua: Advanced Flow
Custom Iterators [4/6]

Here is an actual custom iterator.

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

Lua: Advanced Flow
Custom Iterators [5/6]

We have seen for-in loops that produce multiple values in each iteration (using `pairs`, for example).

```
for x, y, z in fff(3) do
    …
end
```

To allow for this, have the iterator function return multiple values.

```
function fff(n)
    local function iter_func()
        …
        return a, b, c
    end
    …
```

Lua: Advanced Flow
Custom Iterators [6/6]

TO DO

- Write a custom iterator and code that uses it.

Done. See `adv.lua`*.*

This conclues our unit on The Lua Programming Language. However, we will be writing our lexer, parser, and interpreter in Lua. So there is still lots of Lua in our future.

Thoughts on Assignment 2

Thoughts on Assignment 2

This completes the material for Assignment 2.

Three quick notes:

- Exercise A is about running a Haskell program. I will be doing this in the interactive environment (GHCi), and I suggest you learn to run Haskell that way. Start GHCi, type "`:l check_haskell`" (that's an ELL after the colon; also the "`.hs`" is optional), then type "`main`".
- In Exercise B, to test your Lua code, put files `pa2.lua` and `pa2_test.lua` in the same directory. Then run `pa2_test.lua`.
- When writing your code, *get it to run with the test program first!* Then, as you work on it, keep it running with the test program.

Unit Overview
Lexing & Parsing

Our third unit: Lexing & Parsing.

Topics

- Introduction to lexing & parsing
- The basics of lexical analysis
- State-machine lexing
- The basics of syntax analysis
- Recursive-descent parsing
- Shift-reduce parsing
- Parsing wrap-up

After this we will cover The Haskell Programming Language.

Introduction to Lexing & Parsing

Introduction to Lexing & Parsing
What They Are [1/4]

Here are *some* of the things a compiler needs to do.

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

A course on compilers would cover all of the above.

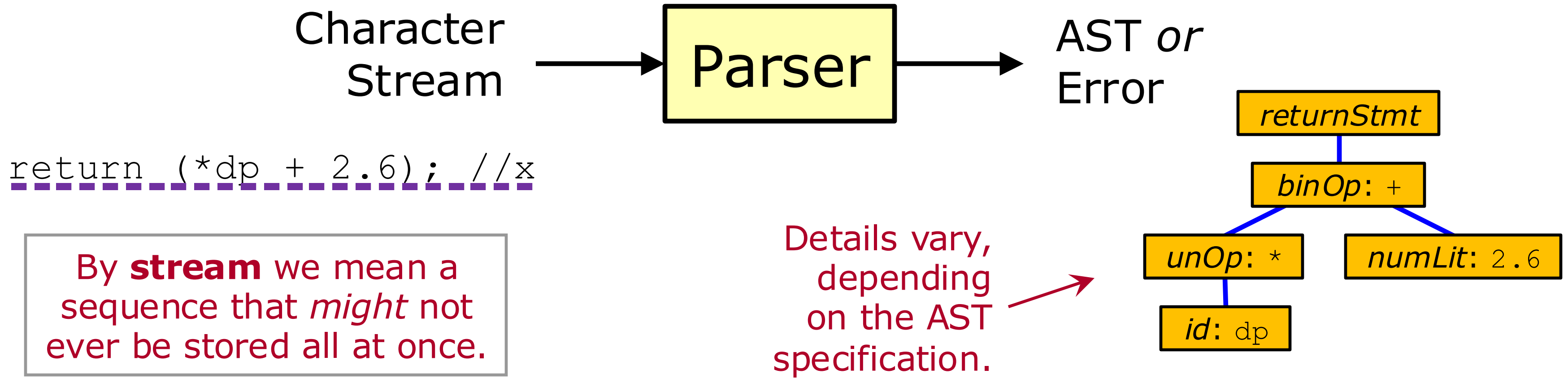
Here, we look at step #1, which is called **syntax analysis**, or **parsing**.

Introduction to Lexing & Parsing
What They Are [2/4]

Parsing: determining whether input is syntactically correct, and, if so, finding its structure.

A software component that does parsing is called a **parser**.

A parser outputs a representation of the structure it finds. This *could* be a parse tree. More commonly, it is an *abstract syntax tree* (*AST*). Such a tree leaves out things like punctuation, which only serve to guide the parser—or human readers.

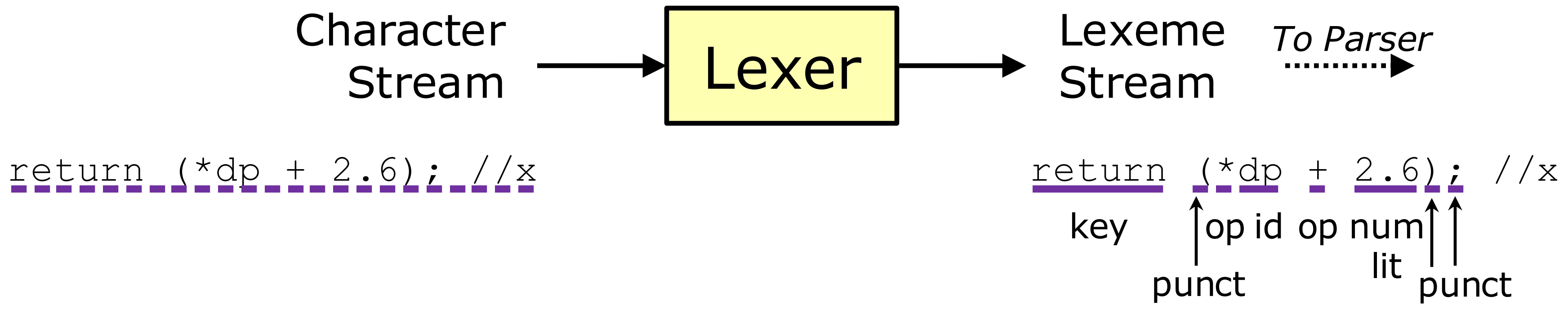

Character Stream
Parser
AST *or* Error
return (*dp + 2.6); //x
By **stream** we mean a sequence that *might* not ever be stored all at once.
Details vary, depending on the AST specification.
returnStmt
binOp: +
unOp: *
numLit: 2.6
id: dp

We will discuss ASTs in detail at another time.

Introduction to Lexing & Parsing
What They Are [3/4]

A preprocessing step is often split off from parsing: **lexical analysis**, or **lexing**. Here input is split into **lexemes** (words, roughly), and the category of each is determined. Things like whitespace and comments are usually skipped.

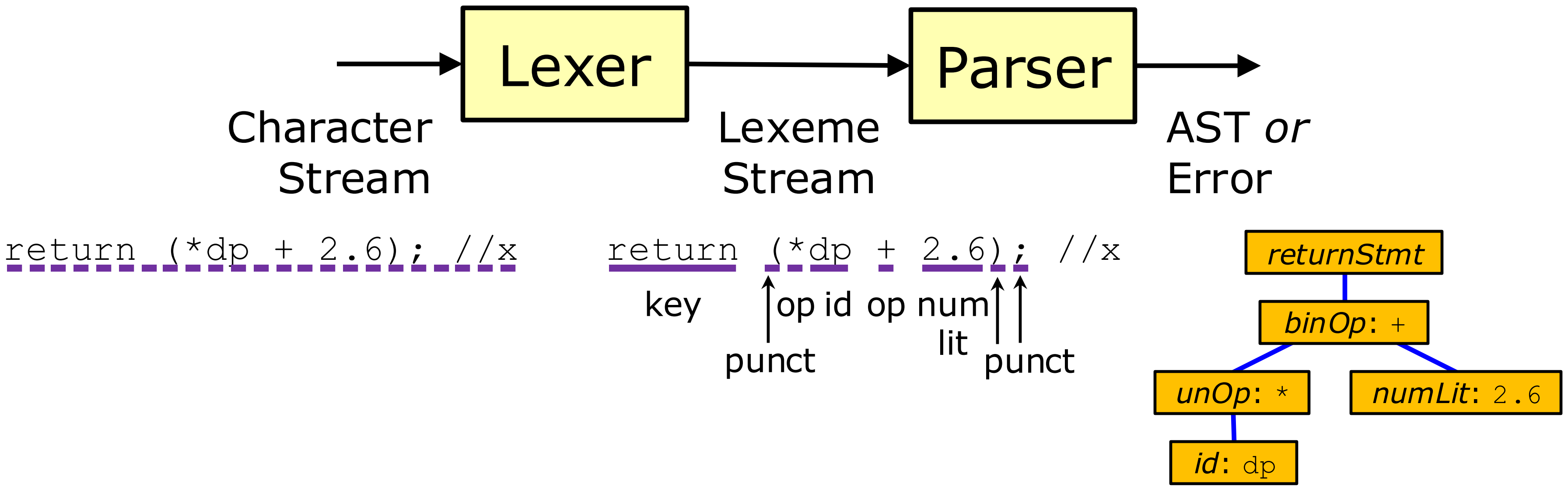
A software component that does lexing is called a **lexer**.



When lexing is split off, the parser takes a lexeme stream as input.

Introduction to Lexing & Parsing
What They Are [4/4]

We will study lexing and parsing as separate steps.



We will write code to do each, in Lua.

Unit Overview
Lexing & Parsing

Topics

- ✓ Introduction to lexing & parsing
- The basics of lexical analysis
- State-machine lexing

Lexical Analysis (Lexing)

- The basics of syntax analysis
- Recursive-descent parsing
- Shift-reduce parsing
- Parsing wrap-up

Syntax Analysis (Parsing)