The Basics of Lexical Analysis
State-Machine Lexing I

CS 331 Programming Languages
Lecture Slides
Wednesday, February 4, 2026

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

© 2017–2026 Glenn G. Chappell

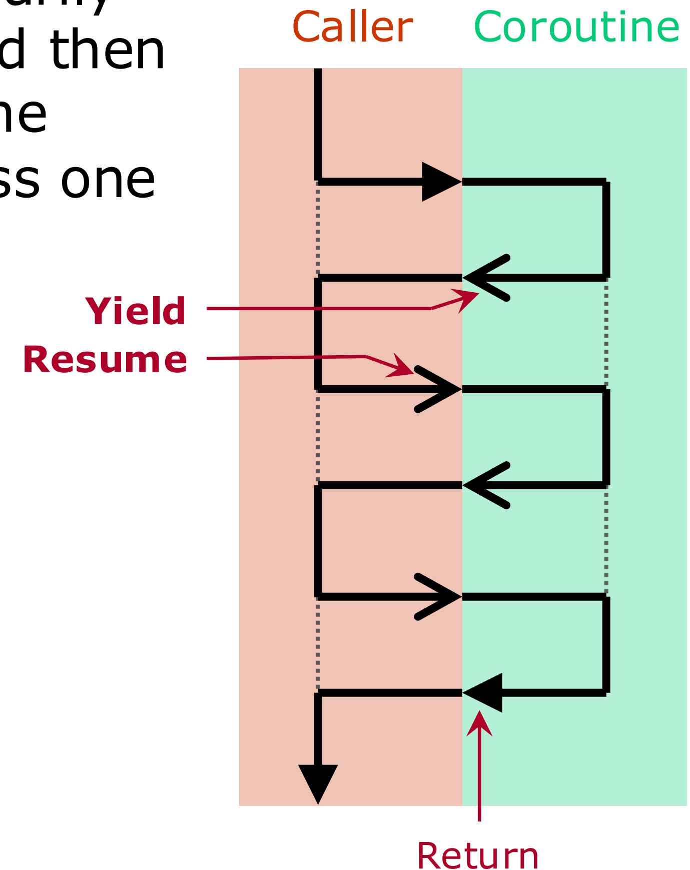
Topics

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

DONE

Review

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).



Coroutines are available in a number of programming languages.

Review
Lua: Advanced Flow [2/2]

In Lua, coroutines are enabled by the standard library `coroutine` module. Call `coroutine.yield` to yield values. Pass a coroutine function to `coroutine.wrap` to get a wrapper function.

A coroutine that yields consecutive integers:

```
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")
```

This code prints:
3 4 5 6 7 8

We also discussed an easy way to turn a coroutine into an iterator, usable by Lua’s for-in loop.

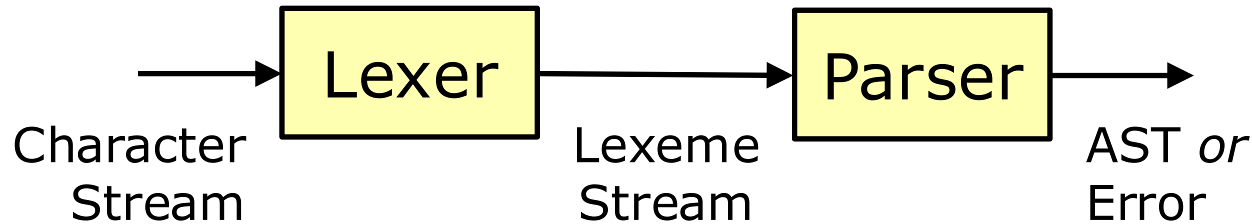
See `adv.lua`.

Unit Overview
Lexing & Parsing

Topics

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

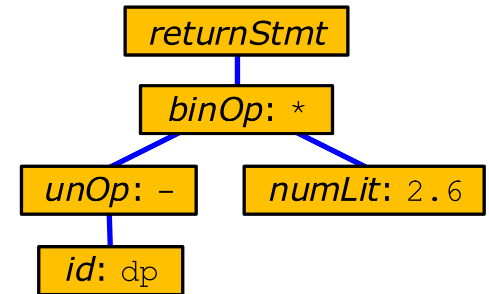
Review
Introduction to Lexing & Parsing



`return (-dp * 2.6) // x`

`return (-dp * 2.6) // x`

key | punct | op | id | op | num lit | punct



Two steps:

- **Lexical analysis** (**lexing**)
- **Syntax analysis** (**parsing**)

The output of a parser is typically an *abstract syntax tree* (*AST*). Specifications of these vary. *We will cover ASTs at another time.*

Unit Overview
Lexing & Parsing

Topics

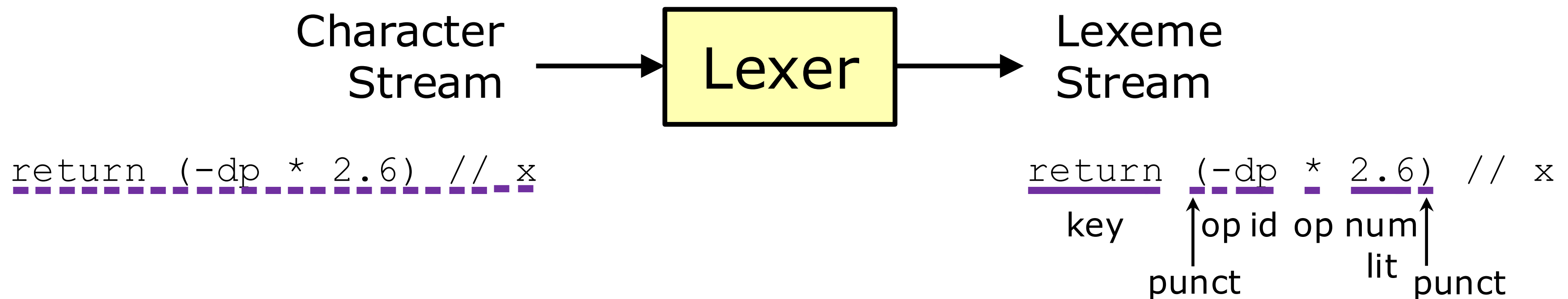
- ✓ Introduction to lexing & parsing
- The basics of lexical analysis
- State-machine lexing I
- State-machine lexing II
- State-machine lexing III
- The basics of syntax analysis
- Recursive-descent parsing I
- Recursive-descent parsing II
- Recursive-descent parsing III
- Shift-reduce parsing
- Parsing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

Lexical Analysis (Lexing): The basics of lexical analysis through State-machine lexing III

Syntax Analysis (Parsing): The basics of syntax analysis through Parsing wrap-up

The Basics of Lexical Analysis

Now we look closer at the first step: lexical analysis, or lexing. A **lexer** reads a character stream and outputs a lexeme stream.



Lexemes are usually classified by **category**.

It is common for each lexeme category to form a regular language. Therefore, what we know about regular languages—including DFAs and regular expressions—is relevant to lexical analysis.

We begin by discussing some common (but not universal!) lexeme categories.

An **identifier** is a name that a program gives to some entity:
variable, class, function, type, namespace, protocol, etc.

In the Swift code below, the identifiers are circled.

```
class MyClass {
    var nn = 7
    var ss = 0
    func computeSumOfSquares() {
        var total = 0
        for ii in 1...nn {
            total += ii*ii
        }
        ss = total
    }
}
```

The Basics of Lexical Analysis
Lexeme Categories [2/5]

A **keyword** is generally an identifier-looking lexeme that has special meaning within a programming language.

In the Swift code below, the keywords are circled.

```
class MyClass {
    var nn = 7
    var ss = 0
    func computeSumOfSquares() {
        var total = 0
        for ii in 1...nn {
            total += ii*ii
        }
        ss = total
    }
}
```

An **operator** is a word that gives an alternate method for making something like a function call. The arguments of an operator are called **operands**.

In the following Swift code, the operators are "`+=`", "`*`", and "`-`". The operands of "`+=`" are "`aaa`" and "`b * -c`".

```
aaa += b * -c
```

The **arity** of an operator is the number of operands it has.

- A **unary** operator has one operand, like "`-`" in the above code.
- A **binary** operator has two operands, like "`+=`" and "`*`" in the above code. A binary operator that is placed between its operands is an **infix** operator.
- A **ternary** operator has three operands. Ternary operators are uncommon. Lua does not have any. However, C, C++, Java and Swift have one: "… `?` … `:` …".

The Basics of Lexical Analysis
Lexeme Categories [4/5]

A **literal** is a representation of a fixed value in source code. The value itself is represented, not an identifier bound to the value, and not a computation whose result is the value.

Swift Literals

Literal	Type
`42`	`Int`
`42.5`	`Double`
`false`	`Bool`
`"goat"`	`String`
`[ 1, 2 ]`	`[Int]`
`( 1, 2 )`	`(Int, Int)`

Lua Literals

Literal	Type
`42.5`	`number`
`nil`	`nil`
`false`	`boolean`
`"goat"`	`string`
`[=[xy]=]`	`string`
`{ 1, 2 }`	`table`

Sometimes a literal is not a single lexeme. These three literals are examples; each consists of 5 lexemes. But every other literal in these tables is a single lexeme.

Lexeme Categories [5/5]

Punctuation is the category for the extra lexemes in a program that do not fit into any of the previously mentioned categories.

Swift punctuation includes braces (`{ }`) and the colon (`:`) in a variable or function declaration.

Lexeme categories mentioned:

- **Identifier**
- **Keyword**
- **Operator**
- **Literal**
- **Punctuation**

The Basics of Lexical Analysis
Reserved Words [1/3]

A **reserved word** is a word that has the general form of an identifier, but is not allowed as an identifier.

Note that, while this is an important concept, *reserved word* is not a lexeme category.

In many programming languages, the keywords and the reserved words are the same.

However, one can imagine a variant of (say) C in which the compiler could distinguish how a word is used, based on its position in the code. Then there could be keywords that are not reserved words. Something like the following might be legal.

```
for (for for = 10; for; --for) ;
```

The Basics of Lexical Analysis
Reserved Words [2/3]

Since the 2011 Standard, C++ has had keywords that are not reserved words; one of these is `override`. This has special meaning when placed after the parentheses in a member-function declaration, but otherwise it is a legal identifier.

So the following is legal C++, by every standard since 2011.

```
class Derived : public Base {
    virtual void override() override;
        // Derived member function named "override"
        // Overrides Base member function "override"
    …
```

The Basics of Lexical Analysis
Reserved Words [3/3]

The programming language Fortran has no reserved words. The following is, famously, legal code in at least some versions of Fortran.

```
IF IF THEN THEN ELSE ELSE
```

On the other hand, there can be reserved words that are not keywords. The Java standard specifies that `const` and `goto` are reserved words. However, neither is a keyword. Thus, these words cannot legally be included in a Java program at all.

We will use our definitions consistently in the class. Be aware, however, that other definitions are used. For example, the C++ Standard does not refer to `override` as a “keyword”.

A lexer outputs a sequence of lexemes. There is usually no need to store the whole sequence. Rather, the lexer can provide get-next-lexeme functionality, which the parser can then use.

There are essentially three ways to write a lexer.

- State machine, entirely in code.
- State machine that uses a table (I do *not* mean a Lua table).
- Using a more advanced method, suitable for writing a parser.

Various software packages—e.g., `lex`—automatically generate code for a lexer, given regular expressions describing the lexeme categories. These use one of the above three methods.

We will write a lexer using the first method. Afterward, I expect that it will be clear how we might have used a table instead.

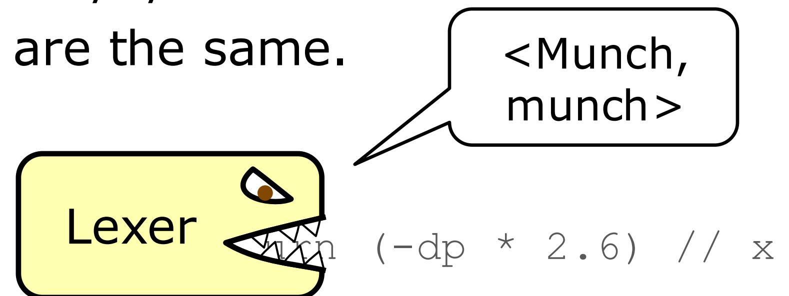
State-Machine Lexing I

State-Machine Lexing I
Introduction [1/2]

We wish to write a lexer, in Lua, for a hypothetical PL. Our lexer will be an entirely hand-coded state machine.

Lexemes are described in the *In-Class Lexical Specification*.

- There might not be any delimiter between lexemes.
- Comments are like multiline C/C++/Swift comments.
- All whitespace outside comments is treated the same.
- Lexemes may be arbitrarily long. *Maximal munch** applies.
- Identifiers are essentially as in Python/C/C++.
- The keywords and reserved words are the same.



***Maximal munch** says that a lexeme is always the longest substring beginning from its starting point that can be interpreted as a lexeme. This applies in some PLs (Lua, C, Java, Python), but not in others (Swift).

State-Machine Lexing I
Introduction [2/2]

Note that our lexeme specification is not universal for all PLs. For example:

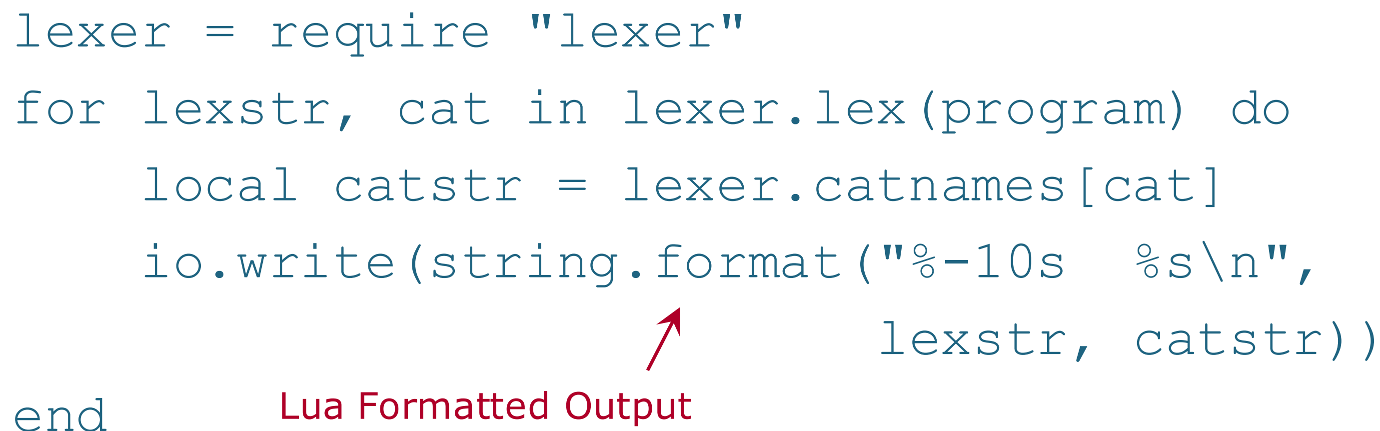
- In Python, Haskell, JavaScript, and Go, a newline may serve as something like an end-of-statement marker; a blank does not.
- In Forth, consecutive lexemes are *always* separated by whitespace.
- Python, Lua, and Haskell use different syntax for comments.
- Haskell allows a different set of characters in identifiers.

State-Machine Lexing I
Design [1/5]

We will write a Lua module `lexer` with the following interface.

- The module has a function `lexer.lex`, which is given a string—the program—and returns an iterator that goes through lexemes.
- The iterator returns 2 values: string and number. The string is the lexeme itself. The number represents the lexeme's category.
- The category is an index for table `lexer.catnames`, whose values will be human-readable string forms of the category names.

So the following prints text & category of all lexemes in `program`.


```
lexer = require "lexer"
for lexstr, cat in lexer.lex(program) do
    local catstr = lexer.catnames[cat]
    io.write(string.format("%-10s  %s\n",
                           lexstr, catstr))
end
```

State-Machine Lexing I
Design [2/5]

The function returned by `lexer.lex` will be a closure. Whatever information is necessary for lexing will be stored in this closure. (Such information is the kind of thing we might store in data members in a Python/C++/Swift object.)

Internally, our lexer will run as a **state machine**.

- A state machine has a current **state**. This might simply be a number. Code that runs as a state machine will need to store this.
- The machine proceeds in a series of steps. At each step, it looks at the state and the input—the current character, at least for now. It then decides what state to go to next.
- Based on the state and the input, our state machine may also make other decisions. Examples might include declaring a lexeme to be complete and sending it back to the caller, or setting the category of a lexeme.

State-Machine Lexing I
Design [3/5]

A skeleton for a lexer is in `lexer.lua`. Our task is to finish this file, turning it into a complete lexer for a hypothetical programming language, whose lexical structure is specified in the *In-Class Lexeme Specification*.

In Assignment 3 you will write a similar lexer, based on a different lexeme specification. This lexer will eventually become part of an interpreter for an actual (not hypothetical) programming language.

I have posted a simple program that uses `lexer.lua`, passing a string ("`program`") and printing the lexemes found. To try it, put `lexer.lua` where it can be imported, and run `use_lexer.lua`. Edit the string `program` in `use_lexer.lua`, if you want.

See `use_lexer.lua`, `lexer.lua`.

State-Machine Lexing I
Design [4/5]

Variables in `lexer.lua`:

- The input is the given string: `program`.
- The index of the next character to read is stored in variable `pos` (which starts at `1`, since this is Lua).
- The state is stored in variable `state`, initialized as `START`.
- We build a lexeme in string `lexstr`, initialized as empty (`""`).
- The category of a complete lexeme is stored in `category`.

When a complete lexeme has been found, set `state` to `DONE`, and set `category` appropriately (`lexer.ID`, `lexer.KEY`, etc.).

State-Machine Lexing I
Design [5/5]

Helper functions in `lexer.lua`:

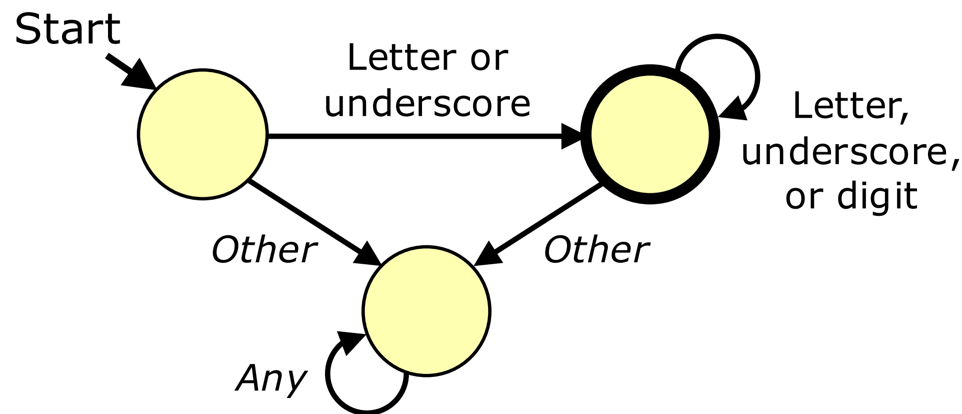
- To add the current character to the lexeme, call `add1()`.
- To skip the current character without adding it, call `drop1()`.
- Lua has no character type. We represent a character as a string of length one. I have written character-testing functions (`isLetter`, `isDigit`, `isWhitespace`, `isIllegal`). Each takes a string. When given a string whose length is not exactly one, each returns `false`.

State-Machine Lexing I
Coding a State Machine [1/4]

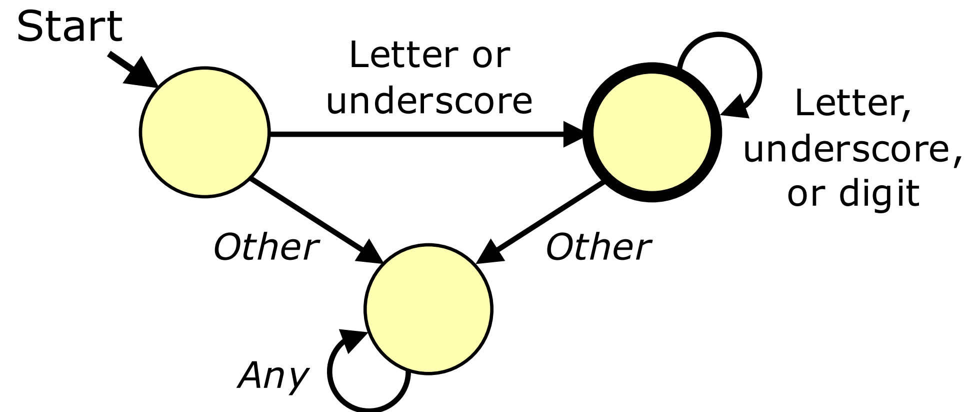
Let's handle **Identifiers** and **Keywords**—ignoring the distinction between the two, for the moment.

An **Identifier-or-Keyword** begins with a letter or underscore (_). Following this are zero or more characters, each of which is a letter, underscore, or digit.

The diagram of a DFA that recognizes an **Identifier-or-Keyword**:



State-Machine Lexing I
Coding a State Machine [2/4]

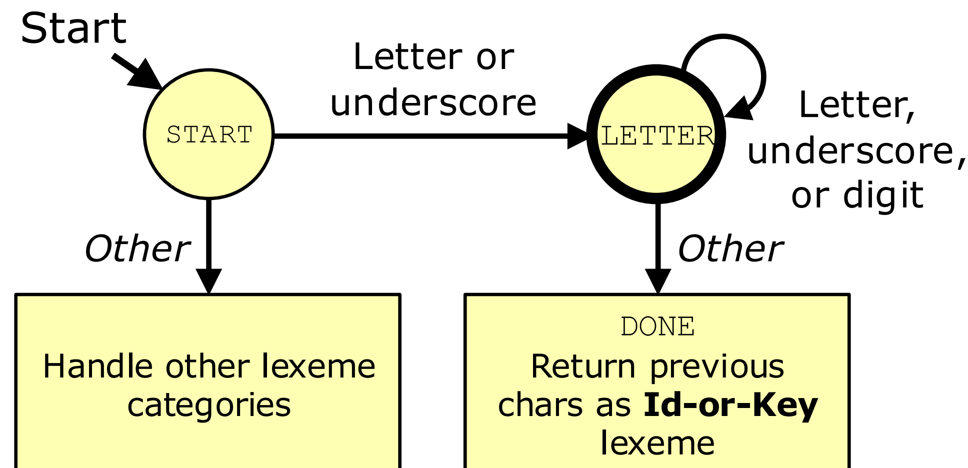


The above DFA is not quite appropriate for what we need to do.

- It determines whether the *entire input* is an **Identifier-or-Keyword**. But we need find these followed by other lexemes.
- We need to handle other categories of lexemes as well.

Note that the two transitions to the lower state are qualitatively different. For our purposes, the one on the left indicates that we have a different lexeme category. The one on the right indicates that an **Identifier-or-Keyword** lexeme is complete.

State-Machine Lexing I
Coding a State Machine [3/4]



Here is an almost-DFA that better expresses our process.

- The right-hand box is the `DONE` state mentioned earlier.
- The left-hand box involves other states that we will discuss later.
- The left-hand circle is the `START` state.

I like to name each state based on a short string that gets the machine into that state. I will call the right-hand circle "`LETTER`". Note that this does *not* mean that we have just read a letter.

State-Machine Lexing I
Coding a State Machine [4/4]

How to differentiate between **Identifier** and **Keyword** lexemes?

We *could* use states to do this, but it would be complicated.

Simpler: when an **Identifier-or-Keyword** lexeme is complete, compare it to each **Keyword**, and then set its category.

TO DO

- Write code to handle **Identifier** and **Keyword** lexemes.
- Test whether this code works.
- As time permits, write other parts of the module.

Done.
See `lexer.lua`*.*

Written in class:

- *Handling of **Identifier**, **Keyword** lexemes, including state* `LETTER`*.*

Written after class:

- *Function* `skipToNextLexeme`
- *Comments on invariants.*