# State-Machine Lexing II

CS 331 Programming Languages
Lecture Slides
Friday, February 6, 2026

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

© 2017–2026 Glenn G. Chappell

## Unit Overview
## Lexing & Parsing

### Topics

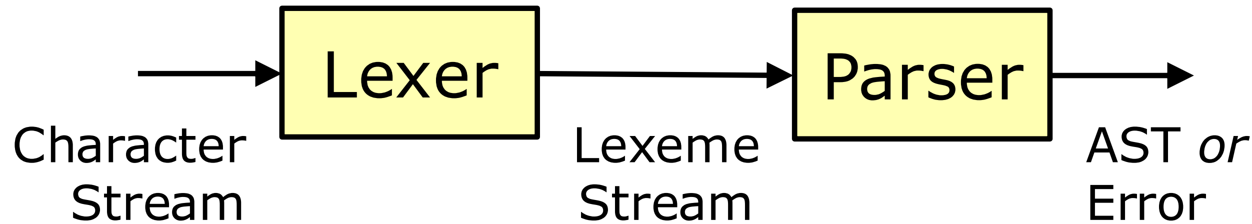
- ✓ Introduction to lexing & parsing
- ✓ The basics of lexical analysis
- ✓ State-machine lexing I
- State-machine lexing II
- State-machine lexing III
- The basics of syntax analysis
- Recursive-descent parsing I
- Recursive-descent parsing II
- Recursive-descent parsing III
- Shift-reduce parsing
-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

Lexical Analysis (Lexing)

Syntax Analysis (Parsing)

# Review

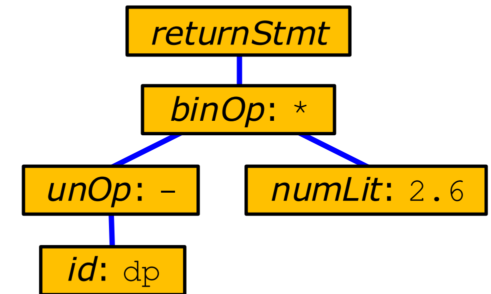
# Review
## Introduction to Lexing & Parsing




`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.

# Review
# State-Machine Lexing I — Design

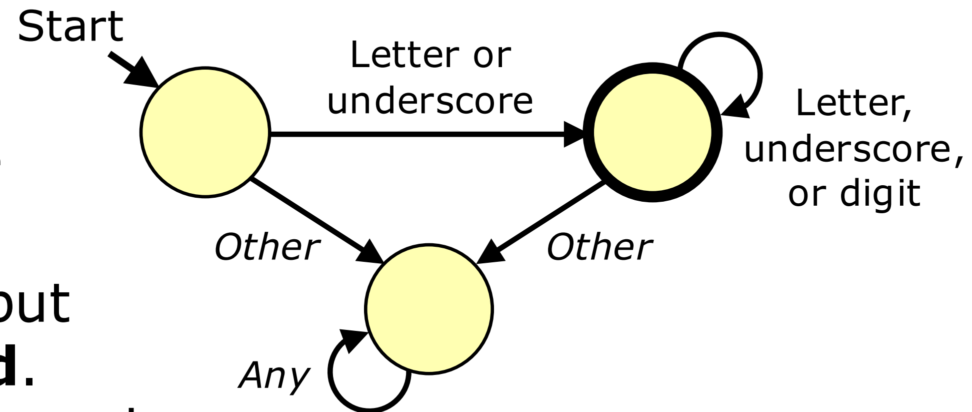
We are writing a Lua module `lexer`, a hand-coded state-machine lexer, based on the *In-Class Lexical Specification*.

Internally, our lexer runs as a **state machine**.

- A state machine has a current **state**. 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, 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, or setting the category of a lexeme.

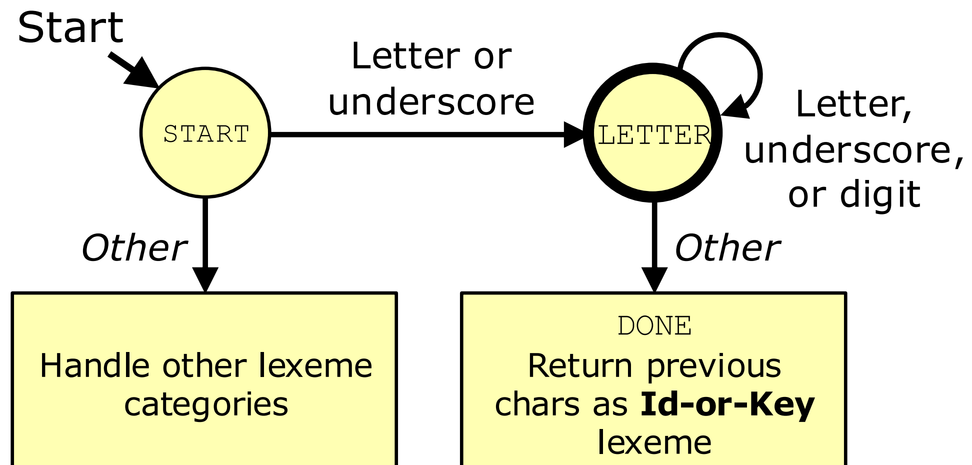
# Review
# State-Machine Lexing I — Coding a State Machine [1/2]

Here is a DFA that recognizes an **Identifier-or-Keyword**. But it is not quite appropriate for what we need to do.




It decides whether the entire input is an **Identifier-or-Keyword**. But we need to find these followed by other lexemes—and to recognize other lexeme categories.

Here is an almost-DFA that better expresses our process. I have named some states.



# Review
# State-Machine Lexing I — Coding a State Machine [2/2]

## Written So Far

- Code to handle **Identifier** and **Keyword** lexemes.
- Function `skipToNextLexeme` (written after class).
- Descriptions of invariants (written after class).

*See* `lexer.lua`*.*

# State-Machine Lexing II

# State-Machine Lexing II
# Issues — Adding a State

As we write a state machine, an important question is *when do we add a new state?*

A guiding principle:

**Two situations can be handled by the same state if they will react identically to all future input.**

For example, if we have read "`a`", then we are in state `LETTER`, since we have read a single letter.

Next, we read "`3`". Are we still in state `LETTER`, or not?

Applying the above principle: yes, we are. Because whatever follows "`a3`" is handled the same as if it followed "`a`". For example, "`a_xq6`" is an identifier, and so is "`a3_xq6`". On the other hand, "`a.`" is not an identifier, and neither is "`a3.`".

# State-Machine Lexing II
# Issues — Invariants [1/2]

We need to be careful about **invariants**: statements that are always true at a particular point in a (properly written) program.

In `lexer.lua`, what do we need to be true about variables—`pos`, in particular—when the construction of a particular lexeme begins? Whatever this is, we need to ensure that it is true when the last code executed before this is finished.

When an invariant is not obvious, it can be good to document it—perhaps in a comment, or perhaps using *assert*. The latter is particularly appropriate when the invariant could easily fail to hold due to incorrectly written code.

# State-Machine Lexing II
# Issues — Invariants [2/2]

Many PLs have a function or function-like thing called “**assert**”. Pass a Boolean expression to it. When it executes, if the expression is false, then the program crashes. Otherwise, the assert does nothing.

Commonly, the above is what happens in a debug build, while in a release build, an *assert* never does anything.

**Swift/Lua**

```
assert(n > 3)
```

**Python**

```
assert n > 3
```

**C++**

```
assert(n > 3);  // Needed: #include <cassert>
```

The expression needs to be one that must be true at this point if the program is written properly—so it needs to be an *invariant*.

# State-Machine Lexing II
# Issues — End of Input

We need to be clear about what happens when we read past the end of the input.

We use the string function `sub` to get single characters out of the input string. This function returns the empty string when it is asked to read past the end of its string.

```
s = "abcdefgh"
z1 = s:sub(2)         -- from position 2 to end: "bcdefgh"
z2 = s:sub(2, 4)      -- positions 2 to 4: "bcd"
z3 = s:sub(2, 2)      -- single character: "b"
z4 = s:sub(91, 99)    -- Past end of string, so empty: ""
```

An empty string gives `false` when passed to a character-testing function, or when equality-compared with a single character. Almost any check about a past-the-end position will give `false`.

# State-Machine Lexing II
## Issues — Skipping

Consider function `skipToNextLexeme`. The job of this function is move `pos` to the beginning of the next lexeme—or the end of the input if there is no next lexeme.

So `skipToNextLexeme` needs to skip comments as well. But it is not enough to skip a single comment. We also need to skip whitespace-comment-whitespace-comment-whitespace, etc.

```
abc
/* comment #1 */   /* comment #2

*/
/* comment #3 */def
```

After reading **this** lexeme [abc], calling `skipToNextLexeme()` needs to take us to the beginning of **this** lexeme [def].

# State-Machine Lexing II
# More Coding a State Machine

TO DO

- Write code to handle other lexeme categories.
- Test whether this code works.

*Done. See* `lexer.lua`*.*

*Written in class:*

- *Handling of* ***NumericLiteral*** *lexemes, including states* `DIGIT`*,* `DIGDOT`*,* `DOT`*,* `PLUS`*,* `MINUS`*.*
- *Handling of* ***illegal*** *characters &* ***Malformed*** *lexemes.*
- *Handling of all* ***Operator*** *lexemes, including state* `STAR`*.*

*Written after class:*

- *Comments on all state-handler functions.*

`lexer.lua` *is now finished (hopefully).*