State-Machine Lexing continued

CS 331 Programming Languages
Lecture Slides
Friday, February 7, 2025

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

Unit Overview
Lexing & Parsing

Topics

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

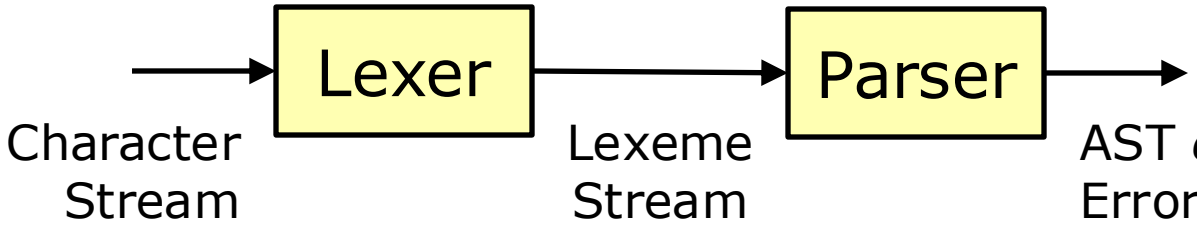
Lexical Analysis (Lexing)

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

Syntax Analysis (Parsing)

Review

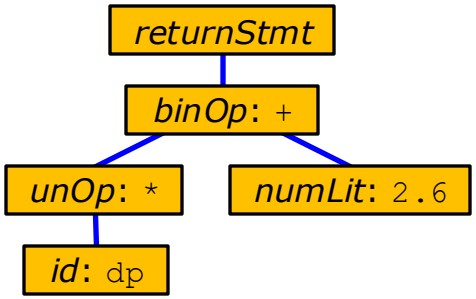
Review
Introduction to Lexing & Parsing




```
return (*dp + 2.6); //x
```

```
return (*dp + 2.6); //x
```

key, op, id, op, num lit, punct, 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.

State-Machine Lexing — Design

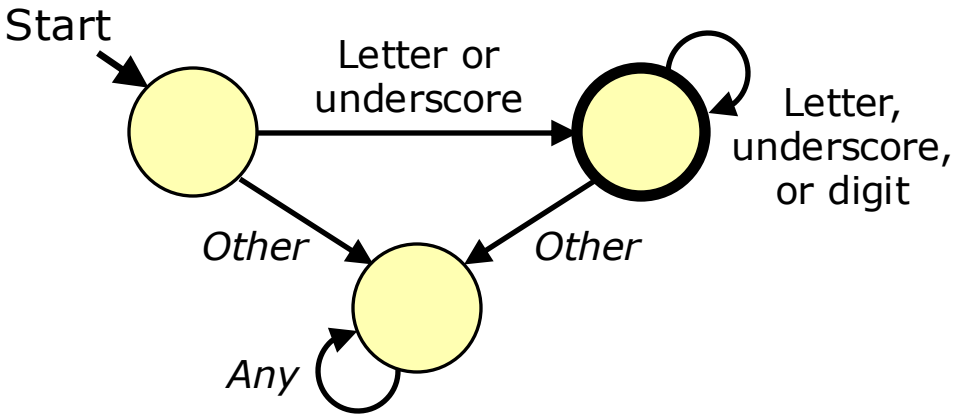
We are writing a Lua module `lexer`, a hand-coded state-machine lexer, based on the *In-Class Lexical Specification*.

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

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

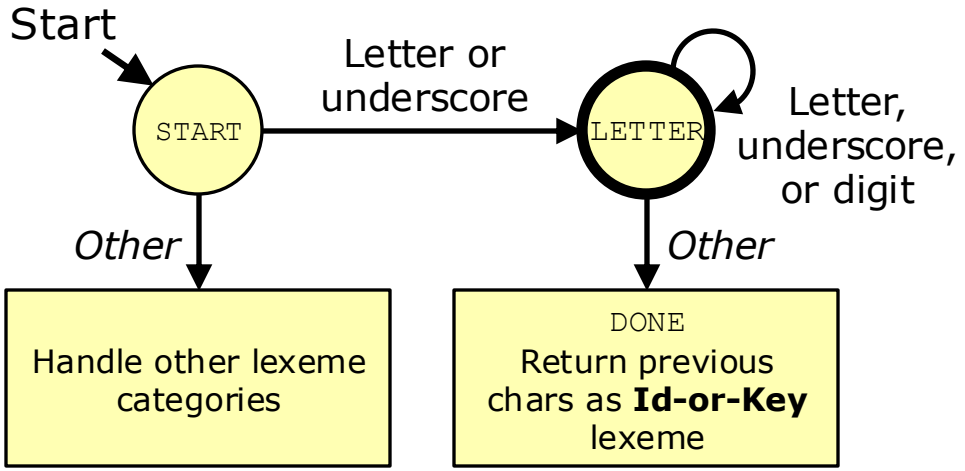
State-Machine Lexing — Coding a State Machine I [1/2]

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



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

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



State-Machine Lexing — Coding a State Machine I [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

continued

State-Machine Lexing
Issues I — 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`", we handle it the same as we would if it followed "`a`". For example, "`a3_xq6`" is an identifier; and so is "`a_xq6`".

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

What do we expect to be true about variables (`pos`, in particular) when our iterator function is *called*? Whatever this is, we need to ensure that it is true when this function *returns*.

When invariants are not obvious, it is a good idea to document them using comments or assertions.

State-Machine Lexing
Issues I — End of Input

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

We use the Lua 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. And an empty string will always result in `false` when passed to a character-testing function, or when equality-compared with any single character. So anything we check about a past-the-end character will be `false`.

```
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: ""
```

State-Machine Lexing
Issues I — Skipping

Consider function `skipToNextLexeme`.

The point of this function is take `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.

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.*

State-Machine Lexing
TO BE CONTINUED …

State-Machine Lexing will be continued next time.