Text-Based Math Calculator Pt II

Lexical Analysis

Recap: What is a Token?

Picking up where we left off with our interactive program.

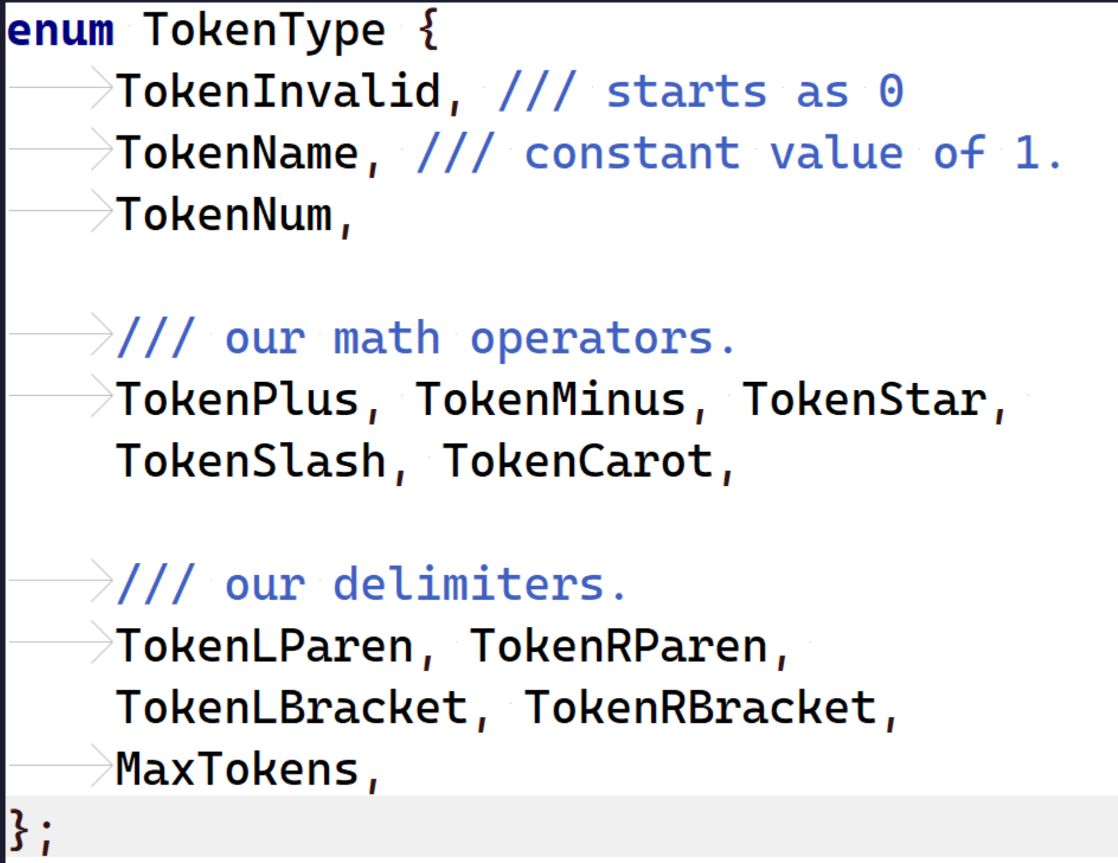
A token is an object describing a set of character(s) and what the meaning of the character(s) are.

Tokens typically hold data like the symbolic value of what the character(s) represents, what part of the string it originated from aka its location like what line, column, etc.

Sometimes tokens hold a copy of the string data or a direct value of the interpreted string data.

How do we give tokens a symbolic meaning?

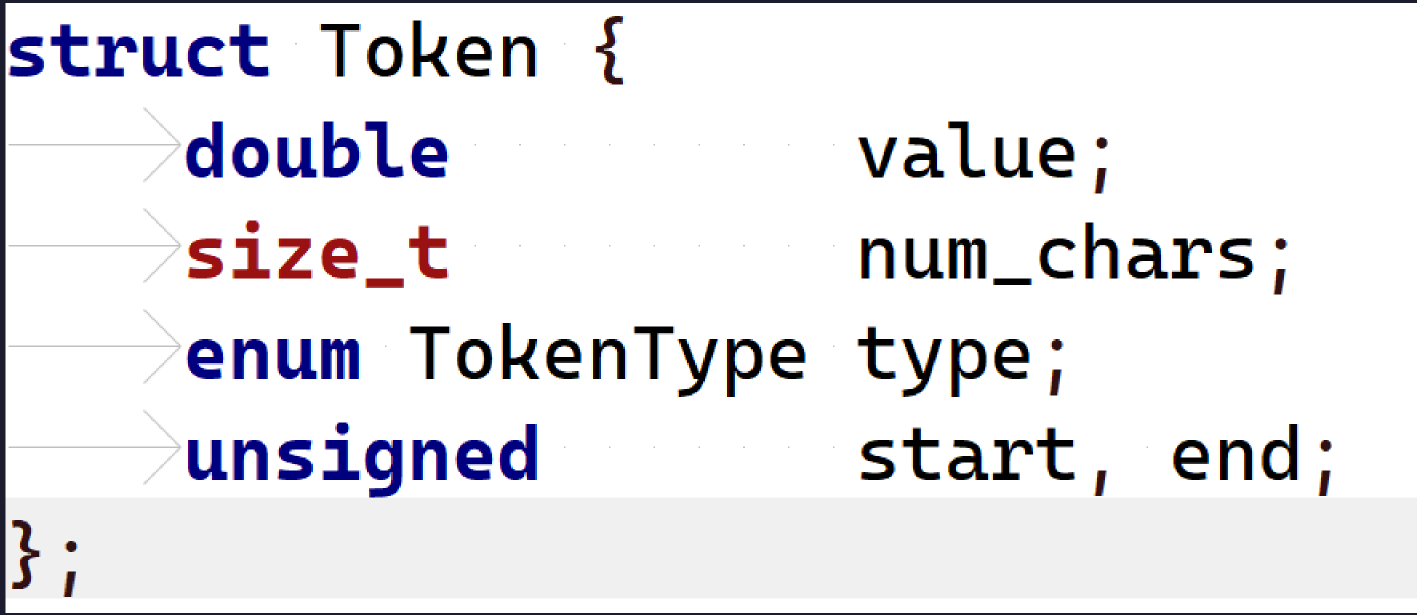
- We start by creating an enumerated list of constants/values to represent the kind of characters we can expect from a math equation.
  - Numbers
    - Can be decimal or non-decimal!
    - Can have a variable length though!
  - Letters/Names of variables/functions.
    - Same thing with numbers, they can be variable length!
  - Add, minus, multiply, division symbols.
  - Delimiters like parentheses or square brackets.



```
enum TokenType {
	TokenInvalid, /// starts as 0
	TokenName, /// constant value of 1.
	TokenNum,

	/// our math operators.
	TokenPlus, TokenMinus, TokenStar,
	TokenSlash, TokenCarot,

	/// our delimiters.
	TokenLParen, TokenRParen,
	TokenLBracket, TokenRBracket,
	MaxTokens,
};
```

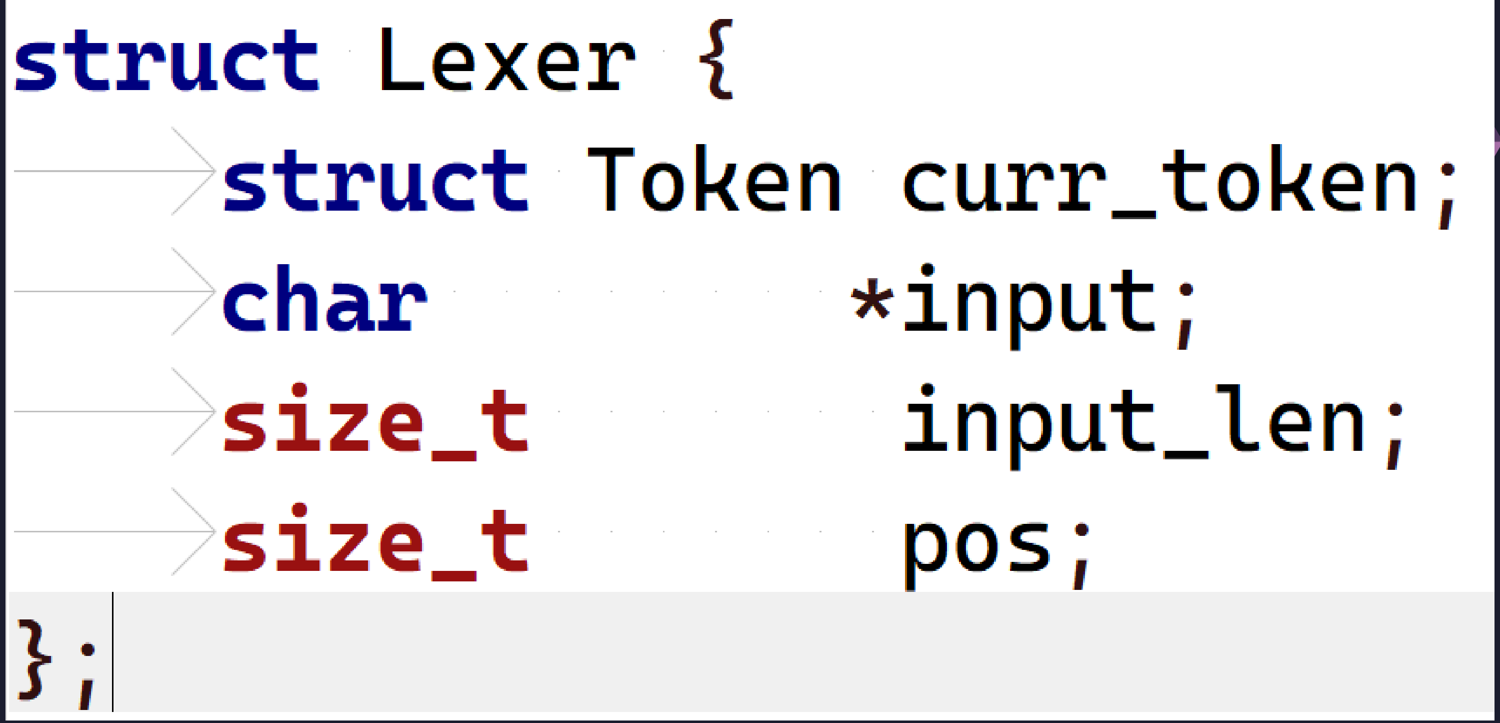
Defining our Token (finally creating objects in C!)

- Tokens don't always have this exact kind of data
- Usually what's needed from the Token is on a program-by-program basis.
- Best for tokens to not record unneeded information.



```
struct Token {
    double          value;
    size_t          num_chars;
    enum TokenType  type;
    unsigned        start, end;
};
```

Creating the Lexer State

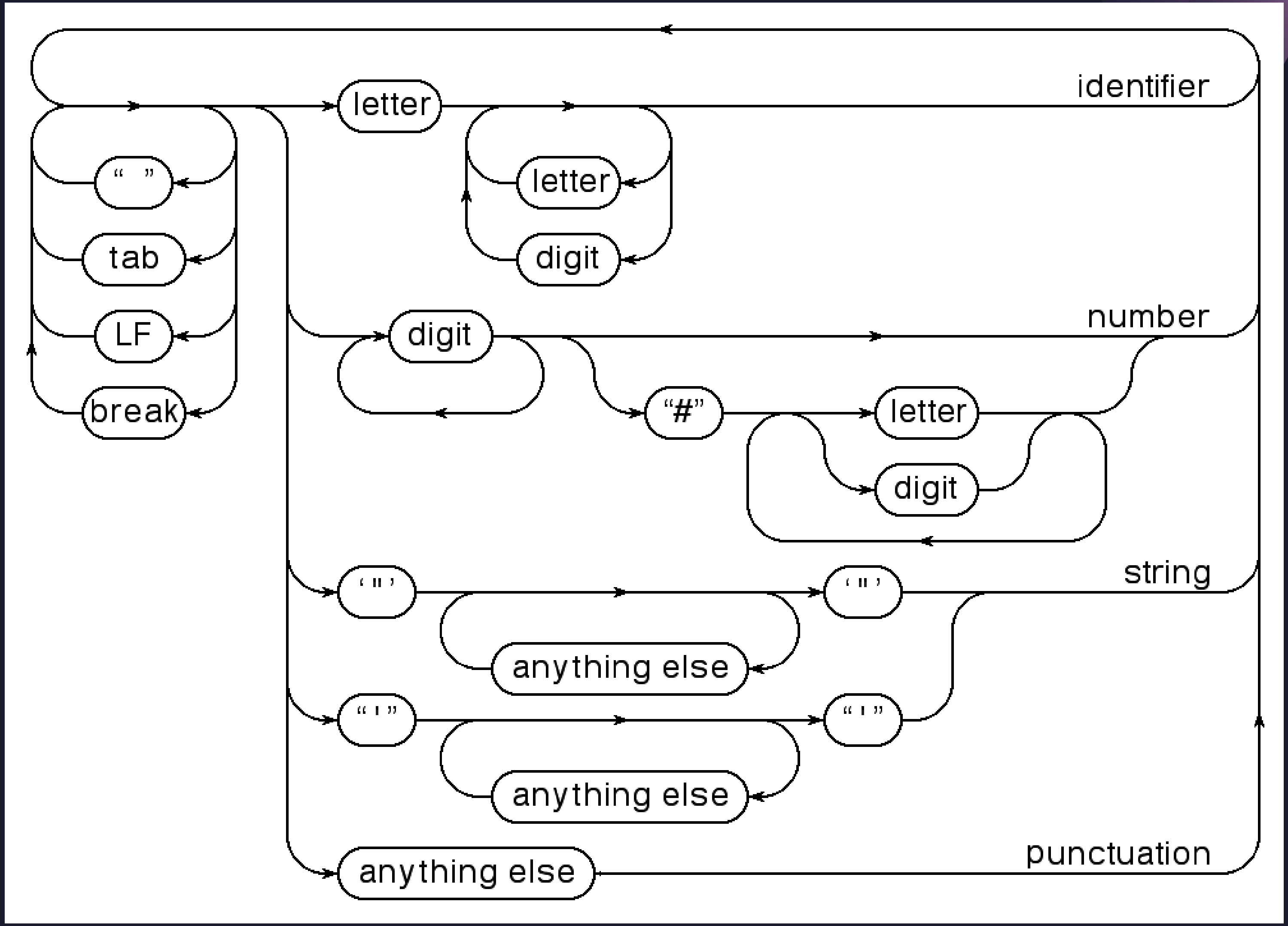


```
struct Lexer {
	struct Token curr_token;
	char         *input;
	size_t        input_len;
	size_t        pos;
};
```

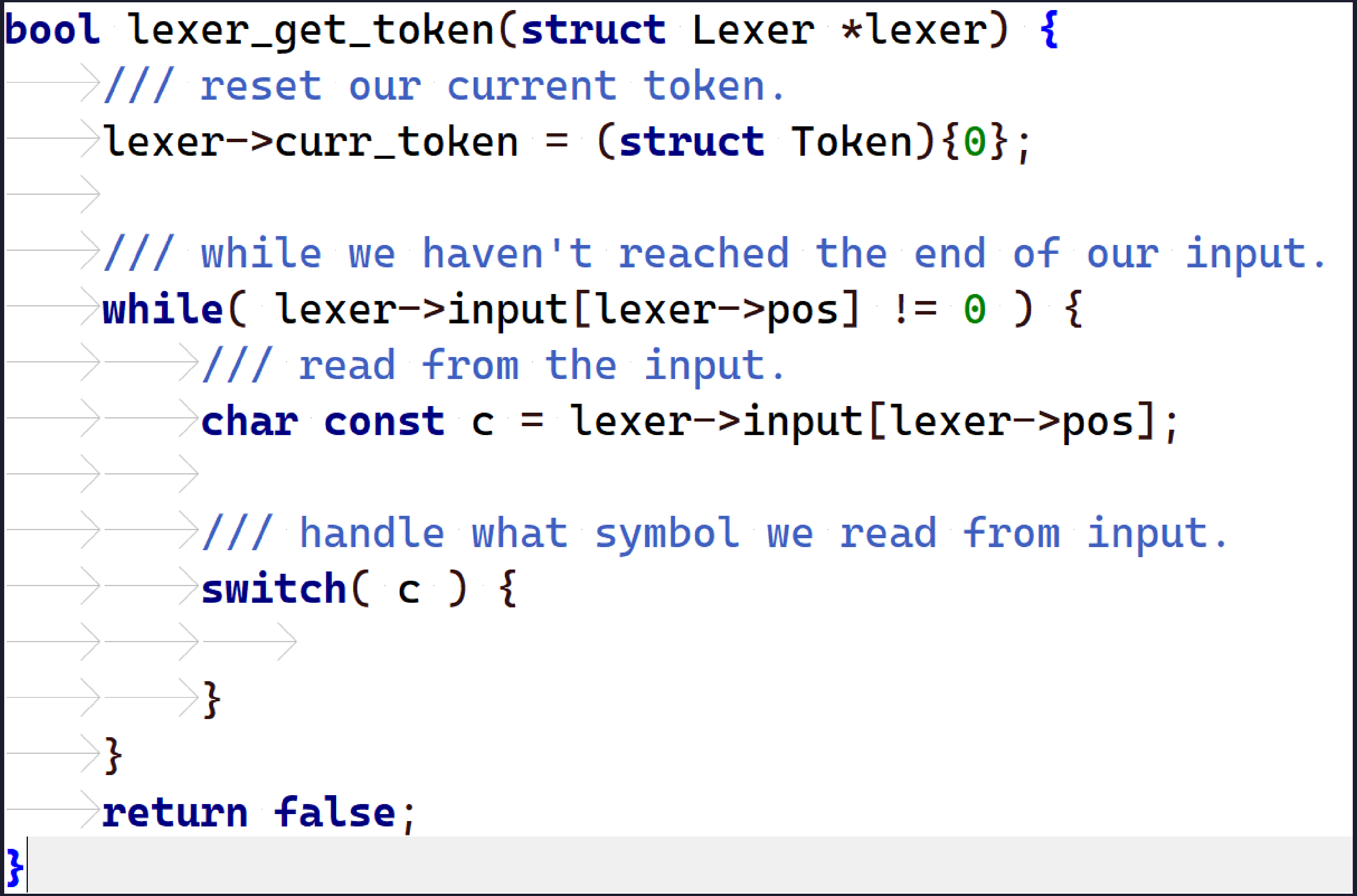
- Typically, a Lexer will hold information on the state of the input like…
    - What the current token is.
    - A way to access the input + how large the input is.
    - Where the lexer is at in terms of the input.
    - Some lexers hold info like the filename, the current line of the input, a token array replacing a current token, and other information as needed by what goals are necessary.
    - For our needs, we don’t need anymore than what we have.

Getting Tokens: The Lexical State Machine

- The most important method is `get_token/getToken`.
- Functions as a state machine:
    - While we haven't reached the end of the input, read a symbol from input, and act accordingly to that symbol.
- Only going to be dealing with numbers and names/identifiers.



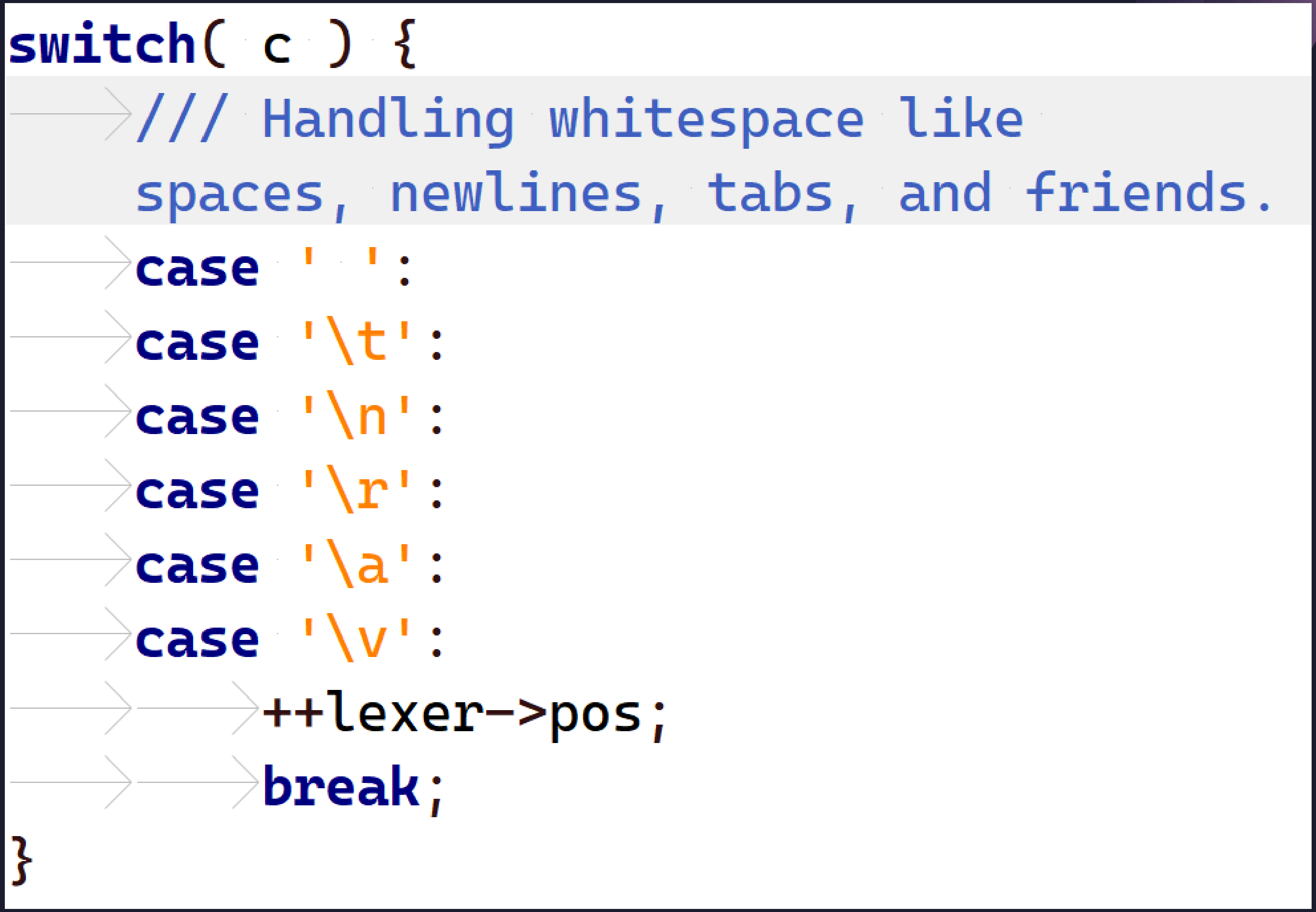
The Shell of the State Machine



```c
bool lexer_get_token(struct Lexer *lexer) {
    /// reset our current token.
    lexer->curr_token = (struct Token){0};

    /// while we haven't reached the end of our input.
    while( lexer->input[lexer->pos] != 0 ) {
        /// read from the input.
        char const c = lexer->input[lexer->pos];

        /// handle what symbol we read from input.
        switch( c ) {

        }
    }
    return false;
}
```

Handling Whitespace

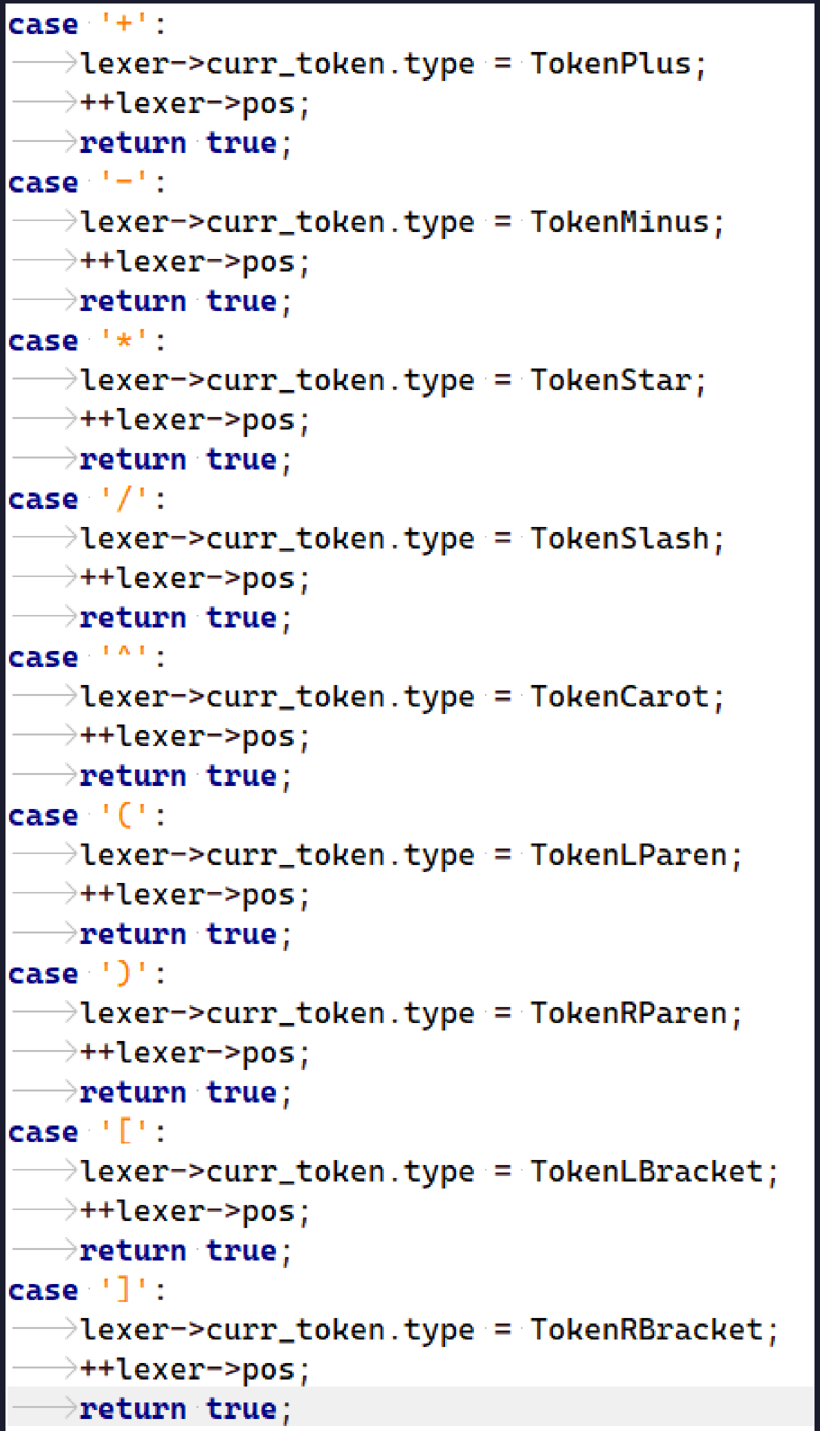
- Whitespace needs to be skipped over. How do we handle that from the state machine?
- We skip over whitespace by simply moving the position data past the whitespace character!



```c
switch( c ) {
	/// Handling whitespace like spaces, newlines, tabs, and friends.
	case ' ':
	case '\t':
	case '\n':
	case '\r':
	case '\a':
	case '\v':
		++lexer->pos;
		break;
}
```

Low-Hanging Fruit:
Handling Operators & Delimiters

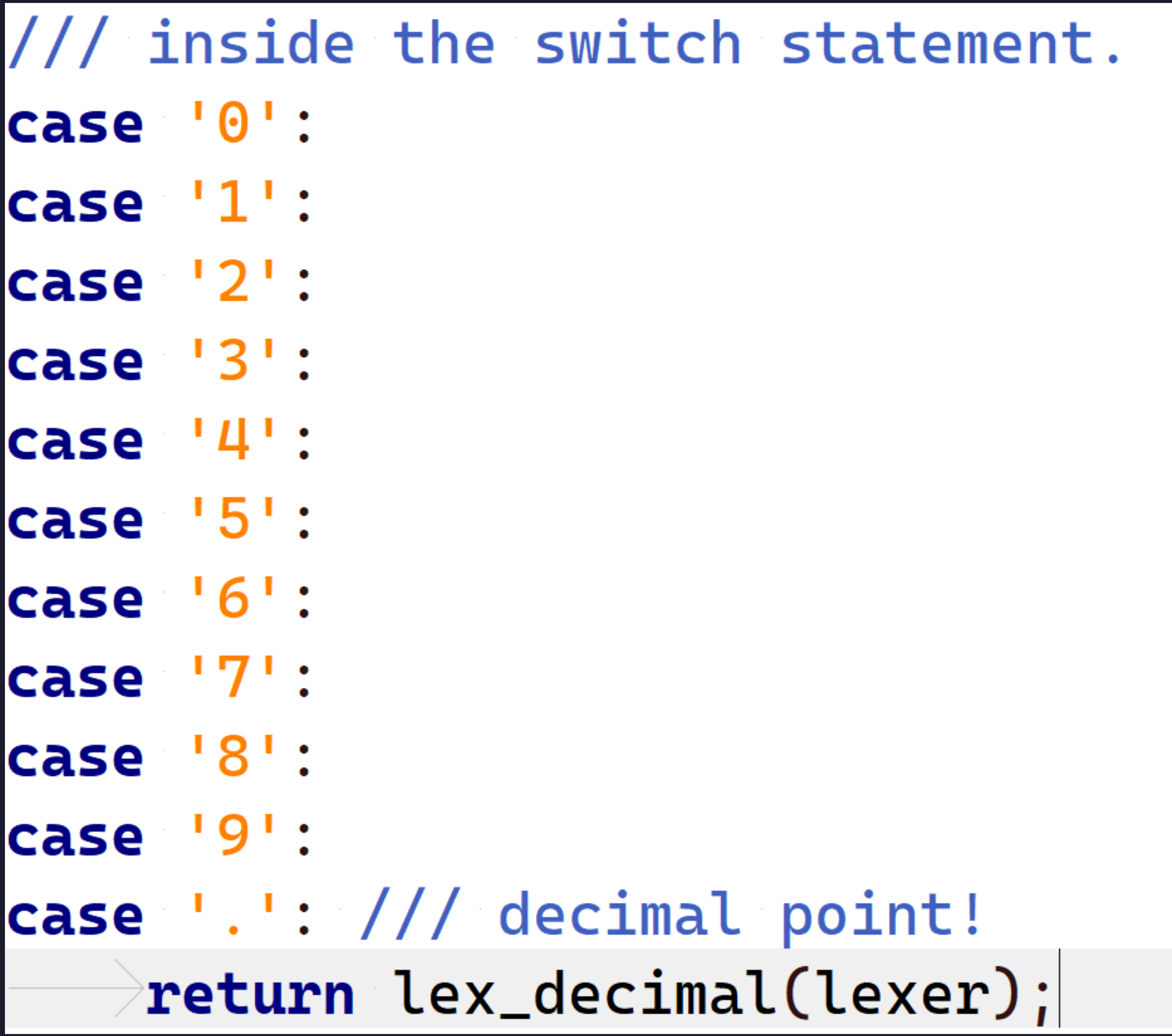
- Handling operators and delimiters like plus sign, parentheses, etc. are rather easy.
- If we do have such of these symbols, we record in our current token that we have the symbol by using its enumerated value.
- After recording our symbol, we skip past it, and return that we were successful.



```
case '+':
	lexer->curr_token.type = TokenPlus;
	++lexer->pos;
	return true;
case '-':
	lexer->curr_token.type = TokenMinus;
	++lexer->pos;
	return true;
case '*':
	lexer->curr_token.type = TokenStar;
	++lexer->pos;
	return true;
case '/':
	lexer->curr_token.type = TokenSlash;
	++lexer->pos;
	return true;
case '^':
	lexer->curr_token.type = TokenCarot;
	++lexer->pos;
	return true;
case '(':
	lexer->curr_token.type = TokenLParen;
	++lexer->pos;
	return true;
case ')':
	lexer->curr_token.type = TokenRParen;
	++lexer->pos;
	return true;
case '[':
	lexer->curr_token.type = TokenLBracket;
	++lexer->pos;
	return true;
case ']':
	lexer->curr_token.type = TokenRBracket;
	++lexer->pos;
	return true;
```

Handling Numbers

- Handling whitespace, single-character operators & delimiters are all pretty easy to do as the last two slides demonstrated.
- Real challenge is handling variable size symbols…
  - Numbers
  - Names/Identifiers
  - Multi-character operators like ==, ->, >>>, &&, etc.
  - **Requires its own state machine to do** (*yes, a state machine in a state machine*)
  - However we already know how easy setting up a state machine is.

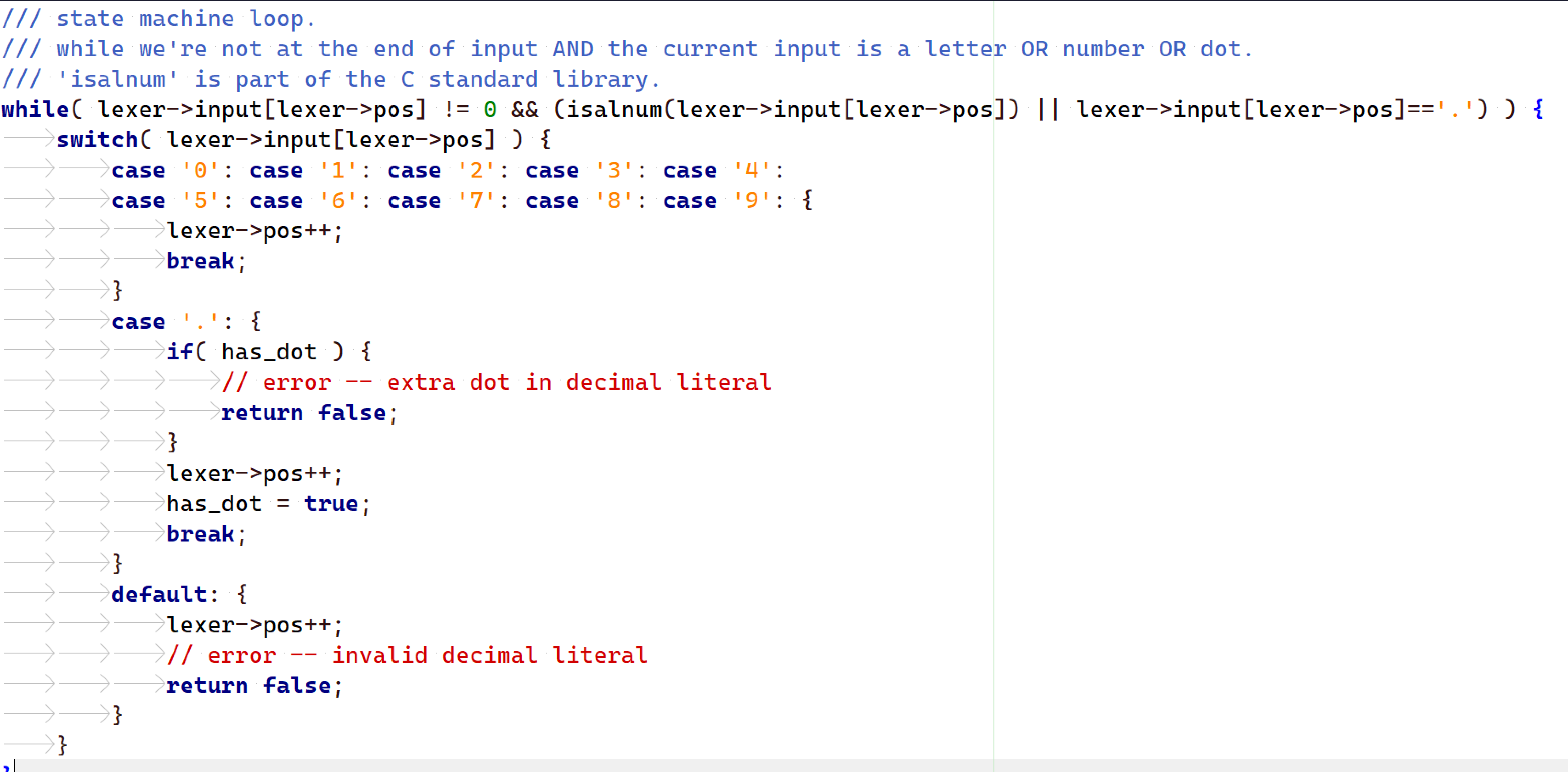


```
/// inside the switch statement.
case '0':
case '1':
case '2':
case '3':
case '4':
case '5':
case '6':
case '7':
case '8':
case '9':
case '.': /// decimal point!
    return lex_decimal(lexer);
```

Handling Numbers Pt 2

```c
bool lex_decimal(struct Lexer *lexer) {
	/// 'has_dot' because we only want to allow ONE decimal point in the number.
	bool has_dot = false;
	
	/// save our old position for token data.
	size_t old_pos = lexer->pos;
	
	/// state machine loop.
	/// while we're not at the end of input AND the current input is a letter OR number OR dot.
	/// 'isalnum' is part of the C standard library.
	while( lexer->input[lexer->pos] != 0 && (isalnum(lexer->input[lexer->pos]) || lexer->input[lexer->pos]=='.') ) {
		switch( lexer->input[lexer->pos] ) {
	}
	/// we didn't get an error, so fill in the data of our current token!
	lexer->curr_token.type = TokenNum;
	
	/// get the actual, usable, printable floating point number.
	lexer->curr_token.value = strtod(&lexer->input[old_pos], NULL);
	
	/// using our saved, old position, we calculate the number of characters of the number.
	lexer->curr_token.num_chars = lexer->pos - old_pos;
	
	/// let's not forget the start and end position of our token!
	lexer->curr_token.start = old_pos;
	lexer->curr_token.end = lexer->pos;
	return true;
}
```

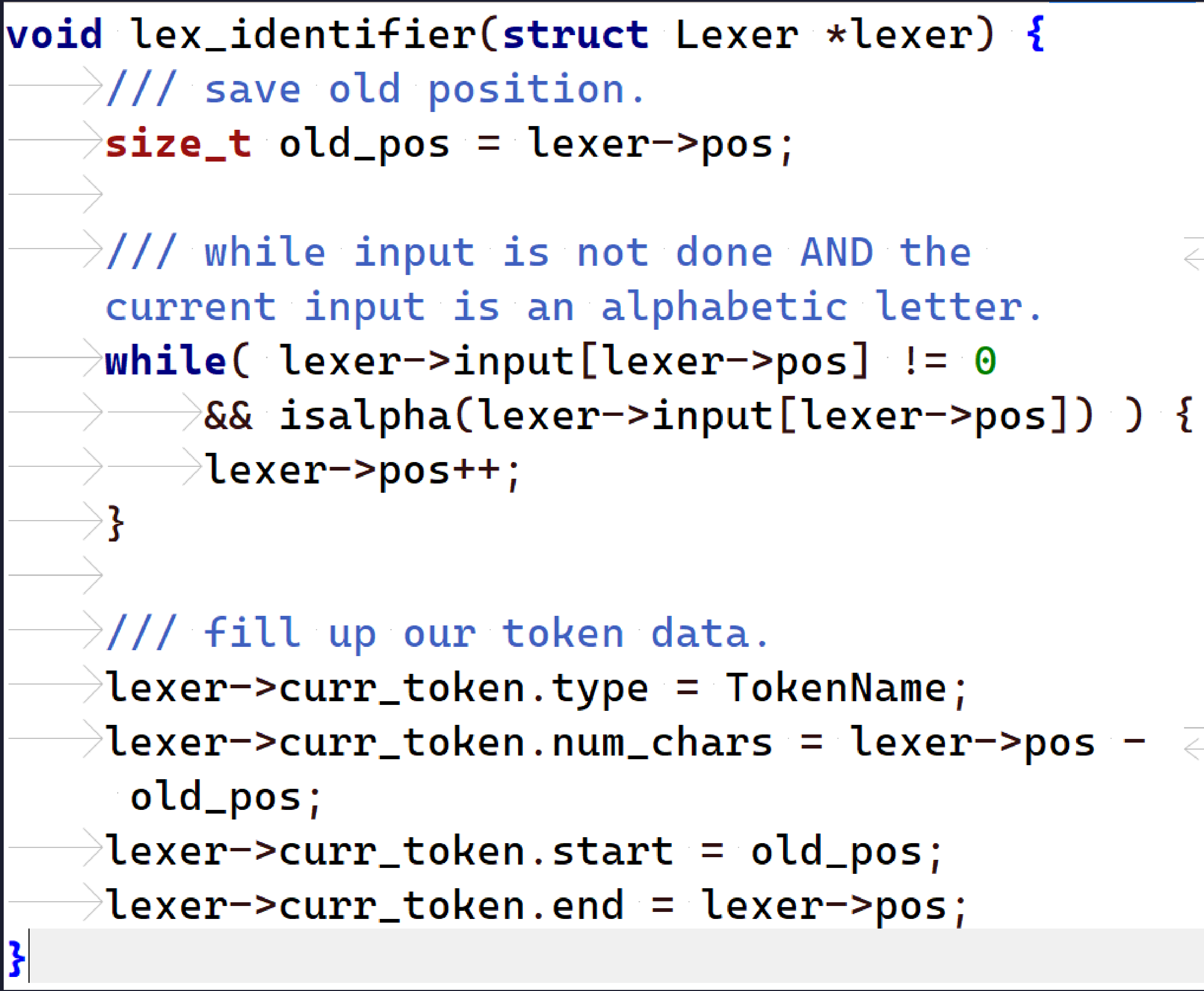
Handling Numbers Pt 3



```c
/// state machine loop.
/// while we're not at the end of input AND the current input is a letter OR number OR dot.
/// 'isalnum' is part of the C standard library.
while( lexer->input[lexer->pos] != 0 && (isalnum(lexer->input[lexer->pos]) || lexer->input[lexer->pos]=='.') ) {
    switch( lexer->input[lexer->pos] ) {
        case '0': case '1': case '2': case '3': case '4':
        case '5': case '6': case '7': case '8': case '9': {
            lexer->pos++;
            break;
        }
        case '.': {
            if( has_dot ) {
                // error -- extra dot in decimal literal
                return false;
            }
            lexer->pos++;
            has_dot = true;
            break;
        }
        default: {
            lexer->pos++;
            // error -- invalid decimal literal
            return false;
        }
    }
```

Handling Names

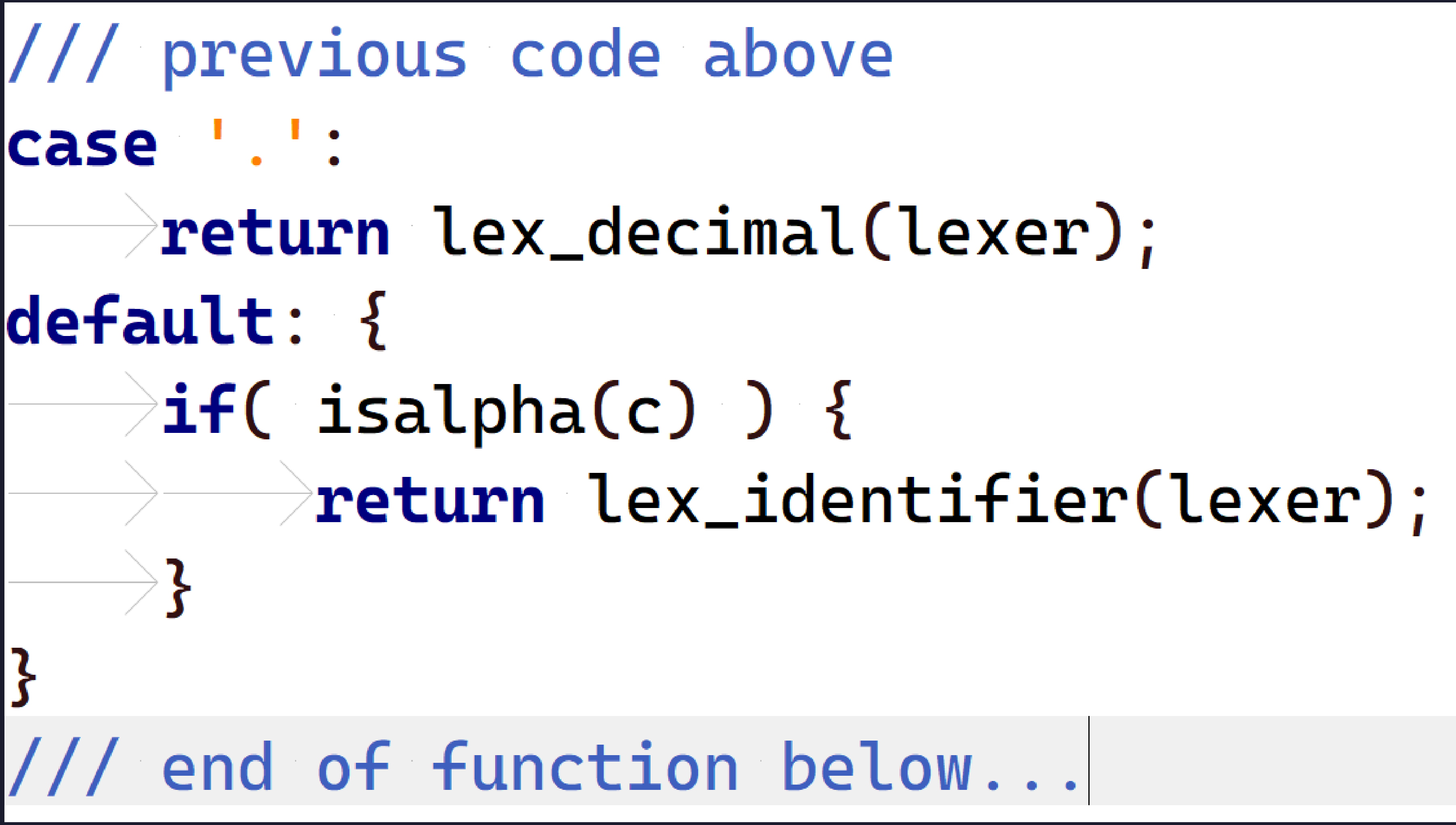
- Handling names is pretty much like handling numbers although much easier since there's a smaller set of possibilities of character placements in the text.



```c
void lex_identifier(struct Lexer *lexer) {
	/// save old position.
	size_t old_pos = lexer->pos;
	
	/// while input is not done AND the
	current input is an alphabetic letter.
	while( lexer->input[lexer->pos] != 0
		&& isalpha(lexer->input[lexer->pos]) ) {
		lexer->pos++;
	}
	
	/// fill up our token data.
	lexer->curr_token.type = TokenName;
	lexer->curr_token.num_chars = lexer->pos -
	 old_pos;
	lexer->curr_token.start = old_pos;
	lexer->curr_token.end = lexer->pos;
}
```

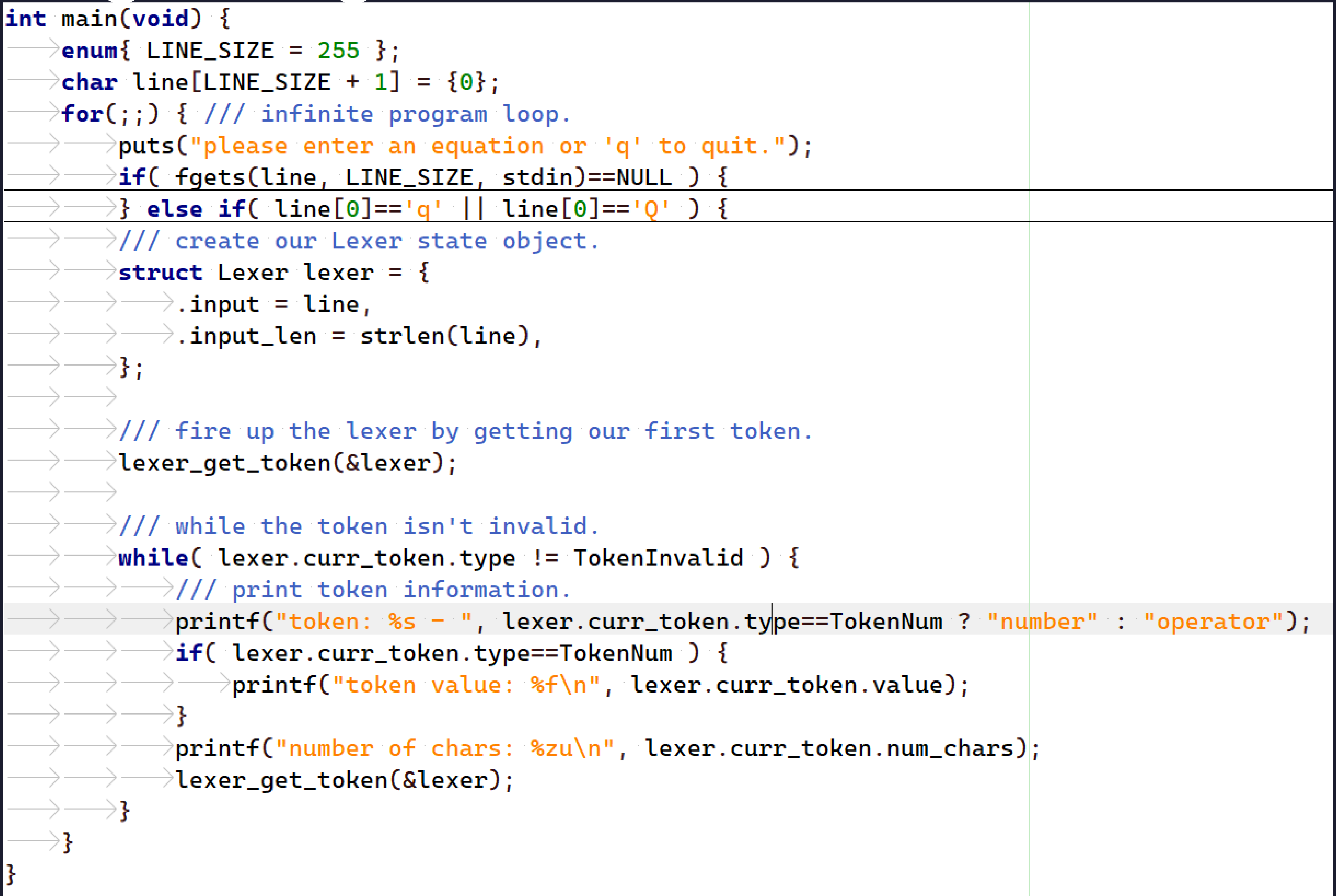
Handling Names Pt 2

- In terms of the main state machine loop. This is where we place how to check for a name variable.
- If it's not an alphabetic character, the `default` will handle that but will be flagged as an error because we're not accounting for it.



```
/// previous code above
case '.':
	return lex_decimal(lexer);
default: {
	if( isalpha(c) ) {
		return lex_identifier(lexer);
	}
}
/// end of function below...
```

Integrating the Lexer code into main



```c
int main(void) {
    enum{ LINE_SIZE = 255 };
    char line[LINE_SIZE + 1] = {0};
    for(;;) { /// infinite program loop.
        puts("please enter an equation or 'q' to quit.");
        if( fgets(line, LINE_SIZE, stdin)==NULL ) {
        } else if( line[0]=='q' || line[0]=='Q' ) {
        /// create our Lexer state object.
        struct Lexer lexer = {
            .input = line,
            .input_len = strlen(line),
        };

        /// fire up the lexer by getting our first token.
        lexer_get_token(&lexer);

        /// while the token isn't invalid.
        while( lexer.curr_token.type != TokenInvalid ) {
            /// print token information.
            printf("token: %s - ", lexer.curr_token.type==TokenNum ? "number" : "operator");
            if( lexer.curr_token.type==TokenNum ) {
                printf("token value: %f\n", lexer.curr_token.value);
            }
            printf("number of chars: %zu\n", lexer.curr_token.num_chars);
            lexer_get_token(&lexer);
        }
    }
}
```

Text-input Calculator Workshop Schedule

- **Intro (talking about Lexical Analysis & Parsing) & Reading file input in C** (*February 2nd, 2024*) [**Done**]
- **Lexically analyzing numbers, names, and operators/symbols** (*Projected February 9th, 2024*) [**Today, Done**]
- **Intro to Parsing theory [Backus-Naur form, Extended Backus-Naur form, Parsing Expression Grammar]** (*Projected February 16th, 2024*)
- **Recursive Descent Parsing and testing** (*Projected February 23rd, 2024*)
- **Adding functions like** ln **and** sin (*Projected March 1st, 2024*)