CSC-363 Lecture 02B
Tokenizing, Continued, Continued

Colin McKinney

28 January 2026

Goal Today / Recap from Last Lecture

So far:

- We've decided on what tokens need to exist.
- We've decided roughly speaking what the tokenizer should produce.
- We've decided what the tokenizer has responsibility for, and what it doesn't.

Today:

- We start to drill down to the actual algorithms and data structures we need.

Studio:

- We drill down further into starting the actual implementation of these data structures and algorithms in Python.

Friday and next week:

- Move on to next stage, parsing, while you work to complete the tokenizer implementation.

Tokenizer Architecture

- Start with a character stream (will be passed to the tokenizer from the main routine)
- Initialize an empty TokenStream (fundamentally a list, but we might want to add more stuff to the object)
- Core loop: while not at end of character stream:
  - Read character(s) from charstream and decide on next token
  - Emit token to TokenStream
- When finished, emit EOF token.

Character Stream Core Architecture

Data needed:

- Source characters: a giant string from the main routine, having stripped out whitespaces, newlines, etc.
- Position marker `pos`, initialized at 0.
- Length of source character string, `sourcelen`, for bounds/EOF checking

Methods needed:

- `eof()->bool` to return if we are or are not at the end
- `peek()->str` that returns character at position `pos`. Does not increment `pos`.
- `advance()` increments `pos`.
- `read()->str` gets char at position `pos`, calls `advance()`, then returns the char.

Each method needs to check `eof()` and handle that case correctly. `read()` and `peek()` should return `''` (empty string) if EOF; `advance()` should do nothing.

Token Class Architecture

- Class `TokenType` will extend `Enum`, and have a numeric code for each token type. Details/rationale later.
- Class `Token` will be the class for tokens. Each token will have:
  - `tokentype` that is an instance of the `TokenType` class
  - `lexeme` that is a string and is the lexeme that gave rise to the token.
  - (Optional) `name: str`, used for int declarations or prints
  - (Optional) `intvalue: int`, used for int literals

Tokenizer Core Architecture

`next_token()` procedure should start with calling `read()` on the character stream. Then:

- If digit: call `readintlit()` to read integer literal
- If `i`: peek next character and
  - Valid letter: int declaration
  - Else: error
- If `p`: (like int declaration!) peek next character and
  - Valid letter: print token
  - Else: error (no printing literals!)
- Valid letter: var ref
- Operator: operator token
- Delimiter: delimiter token
- Else: error

Integer Literals Helper Routine

If in our core loop, `read()` gives us a digit, then we know we're at the start of an integer literal. We can then start a string having that digit as its first character, and:

- While `peek()` returns a digit:
  - Append that char and `advance()`

After loop terminates, we know the next character is not a digit. Emit int literal token, `advance()`, and return to the main loop.

Error Handling

We have a few situations that should throw errors:

- Integer declaration followed by a non-valid variable character
- Print character followed by a non-valid variable character
- Any instance when `peek()` or `read()` get a non-valid symbol

When throwing the error, we should indicate at minimum what the invalid character is. Optionally:

- We could indicate the position in the charstream where it occured (helpful to the human)
- We could also add line/position info to our charstream and preserve newlines (helpful to human, but would require more work)

What Our Tokenizer Should Guarantee

Our tokenizer should pass the following to the next stage:

- Valid token stream
- All tokens in that stream conform to our specifications
- Any character level ambiguity has been dealt with

Our next tasks are to:

- Implement this in Python (starts tomorrow in Studio).
- Begin discussing the next stages of compiling: parsing, semantic actions, code generation, optimization (starts Friday in lecture).