CSC-363 Lecture 02A
Tokenizing, Continued

Colin McKinney

26 January 2026

Tokenizing

Recall that the goal of tokenizing is to take the input program text, as a *character stream*, and output a *token stream*. Running example:

```
i a
i b
b = a + 5
p b
```

As a character stream, this is just `iaibb=a+5pb`. We need to design our token specifications now before trying to implement the tokenizer.

Token Specifications

- Integer declarations: comes from lexeme `i` followed by exactly one letter (excluding `f`, `p`, `i`, `n`).
- Variable reference, single letter lexeme
- Assignment from =
- Delimiters: lpar and rpar
- Binary operations: `+ - * / ^`
- Integer literals: 0, positives only. Ignoring unary minus is a deliberate design decision.
- Print token (lexeme `p`)

On Integer Declarations

We really have two choices when it comes to the integer declarations. Both are reasonable.

- Tokenize `ia` to a single token, `[intdcl; name='a']`
- Tokenize `ia` to two tokens, `[intdcl]` and `varref, name='a']`.

We implicitly used the first option last week, so let's stick with it for consistency. Real languages like C use the second option, pushing the issue to the parsing stage instead.

The Process of Tokenizing

We'll need to keep track of our location in the character stream, much like a “read head." We can advance this read head to look at the next character, or “peek" by looking ahead but not moving the read head. General idea:

- Read character to decide which token type we're dealing with.
- Depending on the type, we might need to read more characters. Examples: `ia` and `67`.
- “Consume" the characters and append the appropriate token to our token stream.

Assuming that we're starting to do a new token:

- Next character is a digit: integer literal; start consuming digits until there are no more digits.
- Next character is `i`: check that following character is a valid letter. If so, emit token; else, error.
- Next char is letter: var reference
- Next char is `p`: print token
- Next char is operator: operator token
- Next char is delimiter: delimiter token
- Whitespace: skip; anything else: error

Example: iaibb=a+5pb

Remaining input	**Action taken**	**Token emitted**
`iaibb=a+5pb`	see `i`, peek `a`	`intdcl(a)`
`ibb=a+5pb`	see `i`, peek `b`	`intdcl(b)`
`b=a+5pb`	see letter	`varref(b)`
`=a+5pb`	see `=`	`assign`
`a+5pb`	see letter	`varref(a)`
`+5pb`	see `+`	`plus`
`5pb`	see digit, consume digits	`intlit(5)`
`pb`	see `p`	`print`
`b`	see letter	`varref(b)`

What we have

- Tokenizer produces a sequence of valid tokens
- Each token has a type and possibly associated data.

What we do NOT have:

- Tokenizing does not check expression structure.
- Tokenizing does not enforce operator precedence or associativity
- Tokenizing does not know if the program “makes sense" (with minor exceptions)