

CSC-363 Lecture 01C

Tokenizing

Colin McKinney

21 January 2026

Tokenizing

Last time we introduced tokenizing briefly. The purpose of this phase is to take the character stream of the source and give it some sort of symbolic structure, to better enable parsing later. Example ac code:

```
1 i a
2 a = a + 5
3 p a
```

This should tokenize to something like

```
[intdcl; name='a'], [intref; name='a'], [assign], [intref; name='a'],  
[intlitt, val='5'], [print], [intref; name='a']
```

The question we want to answer now is how to automate this process.

Terminology

- ▶ Token: a categorized unit
- ▶ Token class/type: which kind of token do we have?
- ▶ Token stream: sequence of tokens. The tokenizer should take a character stream and output a token stream.
- ▶ Lexeme: the actual character sequence that gives rise to a token. Example: `i a` has two characters, and generates the token `[intdcl; name='a']`.

Token types

What things become tokens (in ac)?

- ▶ Keywords: `i`, `p`
- ▶ Identifiers: letters other than `i`, `p`, `n`
- ▶ Literals: integer literals only
- ▶ Delimiters: `(` and `)`
- ▶ Operators: `+` `-` `*` `/` `^`

In C or microC:

- ▶ Keywords: `if`, `then`, `else`, `return`, etc.
- ▶ Delimeters: `{` `}` `;`
- ▶ Boolean operators: `<` `>` `=` `!=` etc.

Regular Languages

Languages are built on alphabets. For example, a language having alphabet $\Sigma = \{a\}$ could be any string with an even number of a s. Or an odd number.

Regular languages over an alphabet Σ are defined recursively as:

- ▶ The empty language \emptyset is regular
- ▶ For each $a \in \Sigma$, the language having exactly one word, $\{a\}$, is regular
- ▶ If A is a regular language, A^* is a regular language. A^* is the Kleene star, which means that any word in A can be repeated as many times (or zero times) as desired. Thus the empty string language $\{\epsilon\}$ is regular.
- ▶ If A and B are regular, so are $A \cup B$ (union) and $A \cdot B$ (concatenation).
- ▶ This list is complete.

Regular Languages and Regular Expressions

Any regular language can be matched by a regular expression, or accepted by either a nondeterministic or deterministic finite automaton (NFA/DFA for short). Examples:

- ▶ `[0-9]` is a regex that defines the language `{'0', '1', ..., '9'}`.
- ▶ `[0-9]+` defines `{'1', '01', '67', '1832', ...}`.
- ▶ `0|[1-9][0-9]*` captures all valid (non-negative) integer literals. How to read this: the vertical bar means OR. So we could have a 0, or we could have anything which has a digit 1-9 followed by zero or more digits 0-9.

Limitations of Regular Expressions

Parentheses make for an interesting problem.

- ▶ We can use a regex to tokenize parentheses, e.g. `\)` or `\(`. After all, a parenthesis is just a symbol, and we need to tokenize to either `lparen` or `rparen`. That's all the tokenization step needs to do.
- ▶ What we **cannot** do with a regex is check whether we have *balanced* parentheses, or if they are used in a syntactically correct way.
 - ▶ But it's OK for now, because the task we're dealing with is tokenizing, not parsing.
 - ▶ Later we will see why regexes are insufficient for the task.
 - ▶ We'll also see what sort of techniques we need instead.

Timeline for Next Week

- ▶ Later today: Problem Set 1 assigned. Due Friday next week.
- ▶ Later today: Programming Assignment 1 assigned. Due a week from Monday.
- ▶ Monday: How tokenizers work generally (but more specific than today).
- ▶ Wednesday: How an ac tokenizer will work.
- ▶ Thursday Studio: Start to implement our tokenizer (basis for PA2).
- ▶ Friday: Moving on from tokenizing to parsing.