Role of Lexer

• Reads characters from the input stream one at a time until a single token is found, and returns it.

• Often called by parser, but can be used generically on its own for other purposes.
Example

```java
if [i == j]
    z = 0;
else
    z = 1;
```

- Goal: partition input into substrings (tokens)
What is a Token?

• A syntactic category

• In English:
  • noun, verb, adjective

• In a programming language:
  • Identifier, Integer, Keyword, Whitespace
Lexers

- Using a set of rules, matches input characters to tokens.

- Tokens include:
  - keywords: `void`, `class`, `this`, `int`
  - punctuation symbols: `=`, `-`, `>=`, `{`
  - identifiers
  - literals: `1.2`, `123`, "hello"

- Lexer will search the patterns in the four categories and return the pattern that matches.
Tokens

- Tokens correspond to a set of strings.

- Identifier: *strings of letters or digits, starting with a letter*

- Integer: *a non-empty string of digits*

- Keyword: “else” or “if” or “begin” or ...

- Whitespace: *a non-empty sequence of blanks, newlines, and tabs.*
What are tokens for?

- Classify program substrings according to role
- Output of lexical analysis is a stream of tokens...
- ... which is input to the parse
- Parser relies on token distinctions:
  - e.g., an identifier is treated differently than a keyword
Designing a Lexical Analyzer: Step 1

• Define a finite set of tokens
  • Tokens describe all items of interest
  • Choice of tokens depends on language, design of parser
Example

• Recall
  \tif (i == j)\n  \t\tz = 0;\n  \telse\n  \t\tz = 1;

• Useful tokens for this expression:
  Integer, Keyword, Relation, Identifier,
  Whitespace, (, ), =, ;

• Note (, ), =, ; are tokens, not characters, here.
Designing a Lexical Analyzer: Step 2

- Describe, using a \textit{pattern}, which strings belong to each token
- Return the value or \textit{lexeme} of the token (\textit{lexeme} is the substring—the actual characters matched).
- Lexer discards characters that don’t belong to a token (e.g., comments, white space, illegal characters) and don’t contribute to parsing
- Reports lexical errors, including:
  - illegal chars
  - unterminated comments
  - unterminated string literals
  - literals too long
Example

• Recall

```c
if (i == j) z = 0; else z = 1;
```

• Token-lexeme groups:
  • Identifier: i, j, z
  • Keyword: if, else
  • Relation: ==
  • Integer: 0, 1
  • (, ), =, ; single character of the same name
Lexer History

- Consider FORTRAN: whitespace is irrelevant

- E.g., VAR1 is the same as VAR1
Example

• Consider
  • DO 5 i = 1,25
  • DO 5 i = 1.25

• The first is DO 5 i = 1 , 25
• The second is DO5i = 1.25

• LtoR, is DO5i a variable or a do statement?
• Can’t tell until after the “,”
Lookahead

• Even simple examples have lookahead issues:
  • i vs if
  • = vs. ==
Lexing C++

- C++ template syntax:
  - `a<b>`
- C++ stream syntax:
  - `cin >> var;`
- C++ binary right shift syntax:
  - `a >> 4;`
- Nested templates:
  - `A<B<C>> D;`
  - `f< a>b > (0);`
- What happens?
Lexical Analysis

• Goal is to partition the string. This is implemented left to right, one token at a time.

• “Lookahead” may be required to decide where one token ends and the next begins.
How To Do This

• Need a way to resolve ambiguities:
  • keyword `if` vs. variables `i` and `f`
  • `==` vs `===`
How To Do This

• A lexer matches the longest lexeme possible. Example:

• ‘E’ might match identifier, but there are other patterns that begin with E that may match:

• ELSE, EXIT, ELSIF, Euphonium (another identifier)
‘E’ Example

string possibilities

- ‘e’ ELSE, EXIT, ELSIF, or ID
- ‘e’ ‘l’ ELSE, ELSIF, or ID
- ‘e’ ‘l’ ‘s’ ELSE, ELSIF, or ID
- ‘e’ ‘l’ ‘s’ ‘e’ ELSE or ID
- ‘e’ ‘l’ ‘s’ ‘e’ ‘2’ must be ID
- ‘e’ ‘l’ ‘s’ ‘e’ ‘2’ ‘ ‘ space is not part of pattern any more
  so return T_ID with the lexeme ‘ELSE2’
Terminating a Pattern

- It isn’t just whitespace
- If you have the input:

  ```
  if(x==3)then
  ```

  - The lexer stops reading when the input can no longer match any available pattern
Terminating a Pattern

- What about:
  - >?
  - >> or
  - >=
Project 1: Lexer

• Matches one token at a time based on an input stream and returns it.

Cool program \[\rightarrow\] text stream \[\rightarrow\] lexer \[\rightarrow\] tokens \[\rightarrow\] parser

token request (some compilers)
Your Lexer will...

- Using a set of patterns given, match input chars to a token
- If token is found, create an instance of the Token class filling in the token type and the actual lexeme
- Along the way, discard characters that don’t belong to a token
- Report lexical errors
What is Online Now

- Assignment Spec
- Tokens Spec (definitions of the different tokens: keywords, ids, punctuation, literals)
- C++ & Java starter files
- C++ & Java support files
The Perfect ECS142 Lexer would:

- recognize tokens
- find lexical errors
  - illegal characters, unterminated strings, etc…
- use correct line #'s
- strip comments
System Details

- Projects are graded on the CSIF.
- Project 1 on the class web site.
Some ideas for starting

Read a character

  If letter
  If #
  If " or '
  If ( (rest of punct chars)
  If whitespace
else error
We can formalize this a bit

- “Start in a start state”
- “Read a character, go to another state.
- When a final state is reached, return.
Regular Languages

- Sounds like a DFA.
- There are several formalisms for specifying tokens
  - *Regular languages* are the most popular.
- Simple and useful theory
- REs and DFAs are interchangeable.