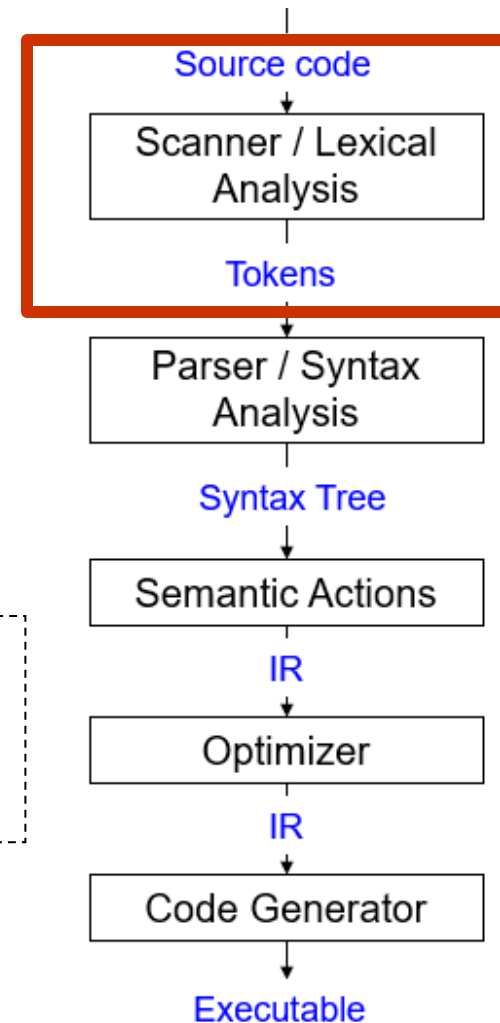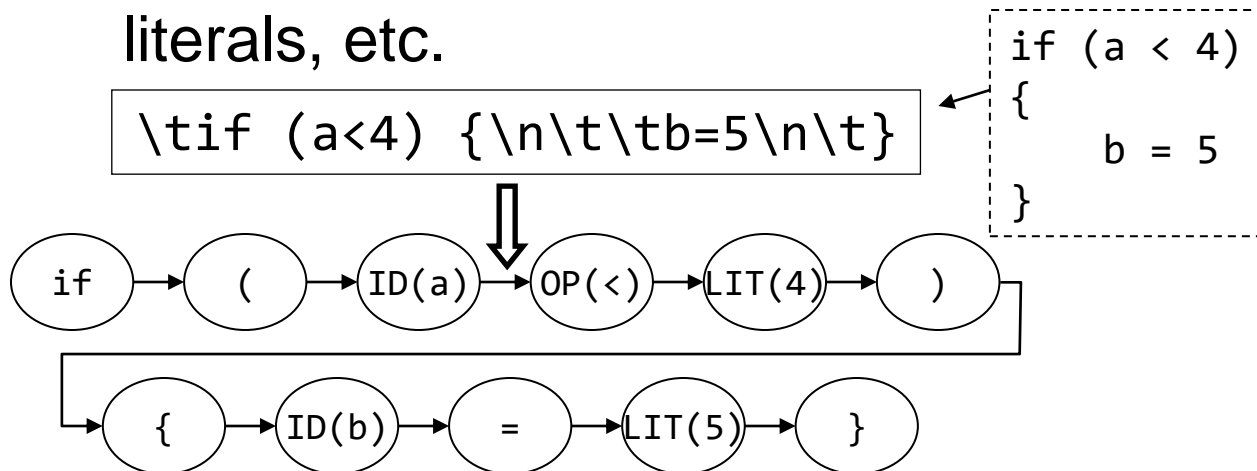# CS323: Compilers
Spring 2023

Week 2: Scanners

Scanner - Overview

- Also called lexers / lexical analyzers

- Recall: scanners
  - See program text as a stream of letters
  - break input stream up into a set of tokens: Identifiers, reserved words, literals, etc.

```
\tif (a<4) {\n\t\tb=5\n\t}
```

```
if (a < 4)
{
    b = 5
}
```


Source code
↓
Scanner / Lexical Analysis
↓
Tokens
↓
Parser / Syntax Analysis
↓
Syntax Tree
↓
Semantic Actions
↓
IR
↓
Optimizer
↓
IR
↓
Code Generator
↓
Executable

2

Scanner - Motivation

- Why have a separate scanner when you can combine this with syntax analyzer (parser)?

    – Simplicity of design
      - E.g. rid parser of handling whitespaces

    – Improve compiler efficiency
      - E.g. sophisticated buffering algorithms for reading input

    – Improve compiler portability
      - E.g. handling ^M character in Linux (CR+LF in Windows)

3

Scanner - Tasks

1. Divide the program text into *substrings or lexemes*
   – place dividers

2. Identify the *class* of the substring identified
   – Examples of predefined categories: Identifiers, keywords, operators, etc.
     - Identifier – *strings of letters or digits starting with a letter*
     - Integer – *non-empty string of digits*
     - Keyword – *"if", "else", "for"* etc.
     - Blankspace - *\t, \n, ' '*
     - Operator – *(,), <, =,* etc.

   – *Observation:* substrings can be categoriezed i.e. follow some pattern

4

Categorizing a Substring (English Text)

- What is the English language analogy for *class*?
  - Noun, Verb, Adjective, Article, etc.
  - In an English essay, each of these classes can have a set of strings.
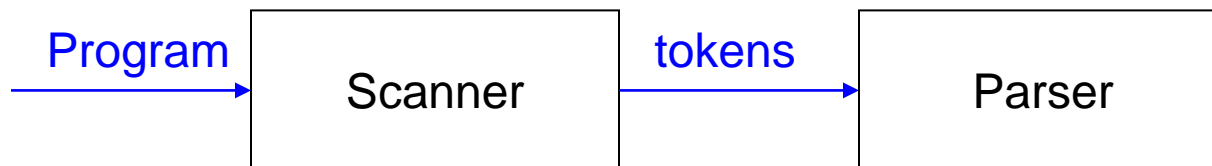 - Similarly, in a program, each class can have a set of substrings.

Exercise

- How many tokens of class *identifier* exist in the code below?

```
for(int i=0;i<10;i++) {
    printf("hello");
}
```

6

Scanner Output

- A token corresponding to each lexeme
  - Token is a pair: <class, value>

A string / lexeme / substring of program text

Program → Scanner → tokens → Parser

```
E.g. int x = 0;                     (Keyword, "int"),
                                    (Identifier, "x"),
                                    ("="),
                                    (Integer, "0"),
                                    (";")
```

Scanners – interesting examples

- Fortran (white spaces are ignored)

    `DO 5 I = 1,25` ⟵——————— DO Loop

    `DO 5 I = 1.25` ⟵——————— Assignment statement

- PL/1 (keywords are not reserved)

    `DECLARE (ARG1, ARG2, . . ., ARGN);`

- C++

    Nested template: `Quad<Square<Box>> b;`

    Stream input: `std::cin >> bx;`

8

Scanners – interesting examples (discussion)

- How did we go about recognizing tokens in previous examples?
    - Scan left-to-right till a token is identified
    - One token at a time: continue scanning the remaining text till the next token is identified...
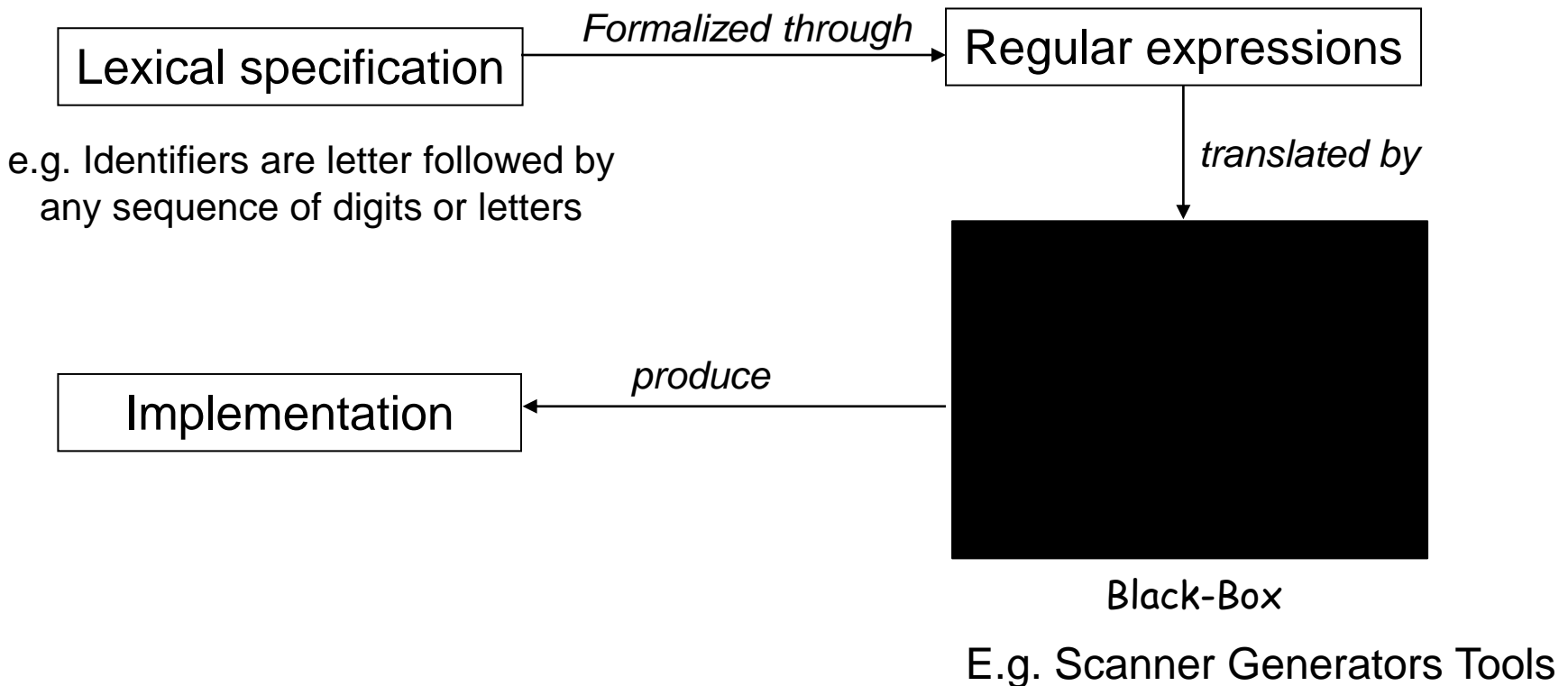   - So on…

We always need to *look-ahead* to identify tokens

….but we want to minimize the amount of look-ahead done to simplify scanner implementation
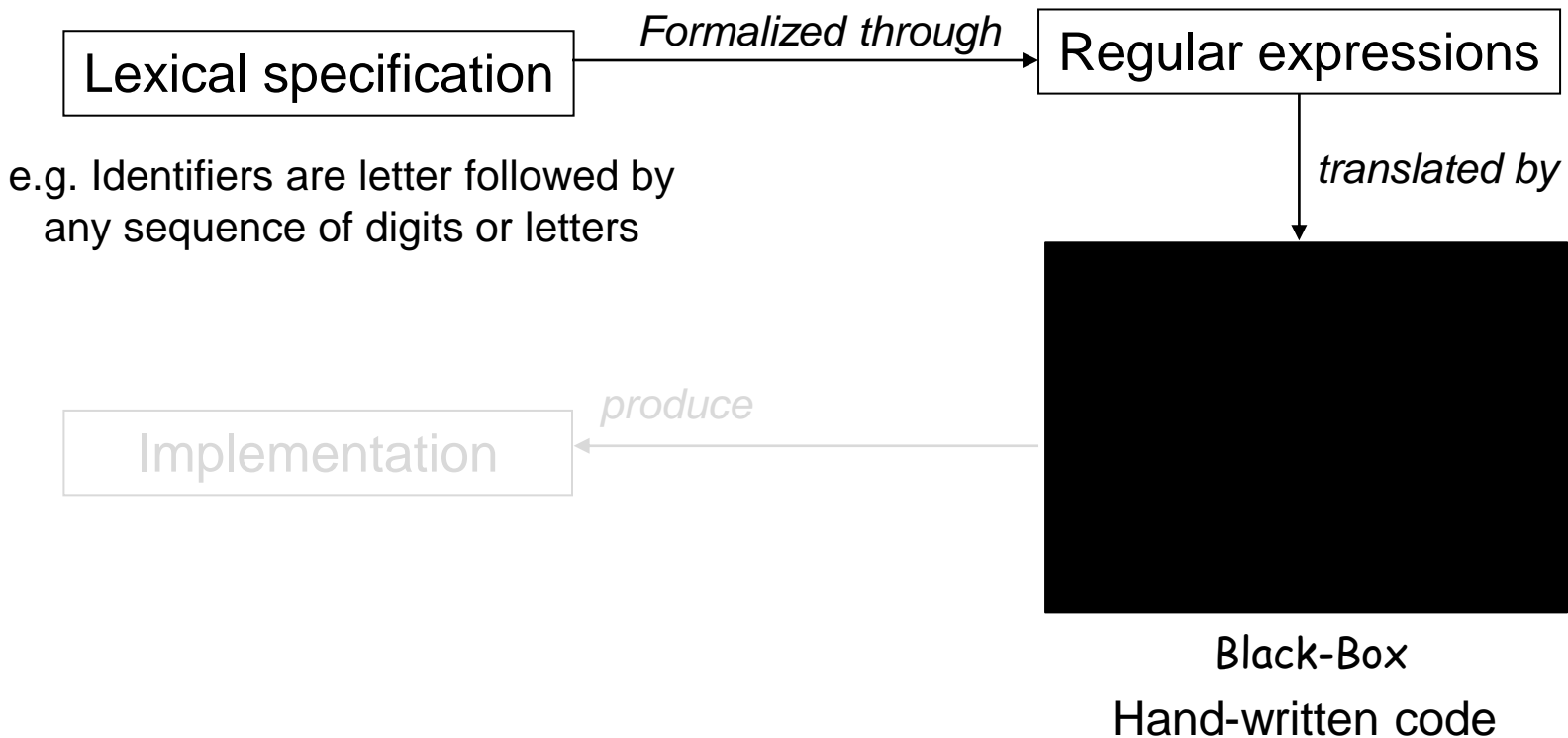
9

Scanners – what do we need to know?

1. How do we define tokens?

    –   Regular expressions

2. How do we recognize tokens?

    –   build code to find a lexeme that is a prefix and that belongs to one of the classes.

3. How do we write lexers?

    –   E.g. use a lexer generator tool such as Flex

Scanner / Lexical Analyzer - flowchart

| Lexical specification | *Formalized through* → | Regular expressions |

e.g. Identifiers are letter followed by
any sequence of digits or letters

translated by

| Implementation | ← *produce* | [Black-Box] |

Black-Box

E.g. Scanner Generators Tools

11

Scanner / Lexical Analyzer - flowchart

Lexical specification ─── *Formalized through* ──→ Regular expressions

e.g. Identifiers are letter followed by
any sequence of digits or letters

translated by

produce

Implementation ←─────────

Black-Box

Hand-written code

Scanner Generators

- Essentially, tools for converting regular expressions into scanners

    - `Lex (Flex)` generates C/C++ scanner program

    - `ANTLR` (ANother Tool for Language Recognition) generates Java program for translating program text (`JFlex` is a less popular option)

    - `Pylexer` is a Python-based lexical analyzer (not a scanner generator). *It just scans input, matches regexps, and tokenizes. Doesn't produce any program.*

13

Regular Expressions

- Used to define the structure of tokens

- Regular sets:

  Informal: a set of strings defined by regular expressions

  Formal: a language that can be defined by regular expressions

  Start with a finite *character set* or *Vocabulary* (V). Strings are formed using this character set with the following rules:

Regular Expressions

- Strings are regular sets (with one element): pi 3.14159
  – So is the empty string: λ (ε instead)

- Concatenations of regular sets are regular: pi3.14159
  – To avoid ambiguity, can use ( ) to group regexps together

- A choice between two regular sets is regular, using |: (pi|3.14159)

- 0 or more of a regular set is regular, using *: (pi)*

- other notation used for convenience:
  – Use Not to accept all strings except those in a regular set
  – Use ? to make a string optional: x? equivalent to (x|λ)
  – Use + to mean 1 or more strings from a set: x+ equivalent to xx*
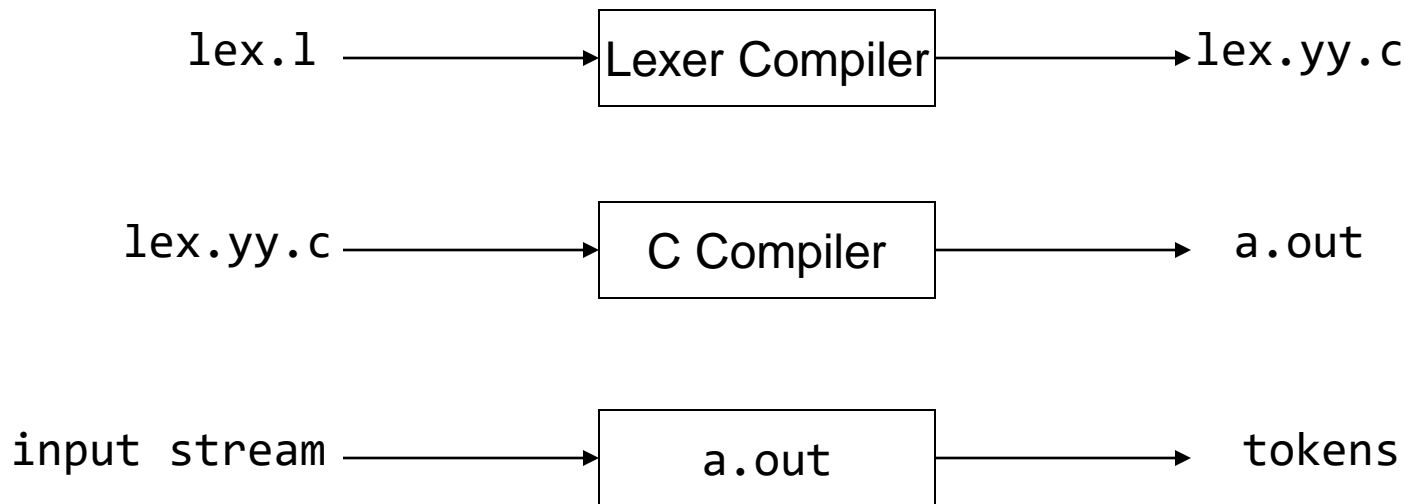 – Use [ ] to present a range of choices: [1-3] equivalent to (1|2|3)

Regular Expressions for Lexical Specifications

- Digit: D = (0|1|2|3|4|5|6|7|8|9)

- Letter: L = [A-Za-z]   alternative definition: [0-9]

- Literals (integers or floats): -?D+(.D*)?

- Identifiers: (_|L)(_|L|D)*

- Comments (as in Micro): //Not(\n)*\n

- More complex comments (delimited by ##, can use # inside comment):

  ## ((#|λ) Not(#))* ##

CS323, IIT Dharwad

Lex (Flex)

- Commonly used Unix scanner generator (superseded by Flex)

- Flex is a domain specific language for writing scanners

- Features:

    - Character classes : define sets of characters (e.g., digits)

    - Token definitions : regex {action to take}

Lex (Flex)

lex.l ⟶ | Lexer Compiler | ⟶ lex.yy.c

lex.yy.c ⟶ | C Compiler | ⟶ a.out

input stream ⟶ | a.out | ⟶ tokens

18

Lex (Flex)

- Format of lex.l (3 parts separated by %%)

format: **name definition**

Declarations ←

e.g. DIGIT [0-9]

Refer to DIGIT here
using {} braces {DIGIT}
expands to ([0-9])

%%

Translation rules

format: **pattern action**

e.g. {DIGIT}+ {printf("INTLITERAL");

%%

User code mentioned here copied as is to lex.yy.c

Auxiliary functions

Example: Lex (Flex)

```
DIGIT       [0-9]
ID          [a-z][a-z0-9]*


%%

{DIGIT}+    {
             printf( "An integer: %s (%d)\n", yytext,
             atoi( yytext ) );
        }


{DIGIT}+"."{DIGIT}* {
             printf( "A float: %s (%g)\n", yytext,
             atof( yytext ) );
        }


if|then|begin|end|procedure|function {
             printf( "A keyword: %s\n", yytext );
        }


{ID}        printf( "An identifier: %s\n", yytext );
```

slide courtesy: Milind Kulkarni

Lex (Flex)

- The order in which tokens are defined matters!

- Lex will match the longest possible token

  - "ifa" becomes ID(ifa), not IF ID(a)

- If two regexes both match, Lex uses the one defined first

  - "if" becomes IF, not ID(if)

- Use action blocks to process tokens as necessary

  - Convert integer/float literals to numbers

  - Remove quotes from string literals

21

Demo

Documentation

- [Flex (manual web-version):](#)
- [Lexical Analysis With Flex, for Flex 2.6.2: Top (westes.github.io)](#)
- [Lex - A Lexical Analyzer Generator (compilertools.net)](#)
- [ANTLR](#)