CS406: Compilers

Spring 2022

Week 2: Overview (winding up), Scanners

Design Considerations

- Compiler and programming language designs influence each other
  - Higher level languages are harder to compile
    - More work to bridge the gap between language and assembly
  - Flexible languages are often harder to compile
    - Dynamic typing (Ruby, Python) makes a language very flexible, but it is hard for a compiler to catch errors (in fact, many simply won’t)
  - Influenced by architectures
    - RISC vs. CISC

Programming Languages and Real-world Usage

- Why are there so many programming languages?
- Why are there new languages?
- What is a good programming language?

Programming Languages and Real-world Usage

- Why are there so many programming languages?
  - Distinct often conflicting requirements of the application domain

Scientific Computing	Floating-Point Arithmetic, Parallelism Support, Array Manipulation	FORTRAN
Business Applications	No data loss (persistence), Reporting capabilities, Data analysis tools	SQL
Systems Programming	Fine-grained control of system resources, real-time constraints	C/C++

Programming Languages and Real-world Usage

- Why are there new languages?
  - To fill a technology gap
    - E.g. arrival of Web and Java
    - Java’s design closely resembled that of C++

Training a programmer on a new programming language is a dominant cost

- Widely-used languages are slow to change
- Easy to start a new language

Programming Languages and Real-world Usage

- What is a good Programming Language?

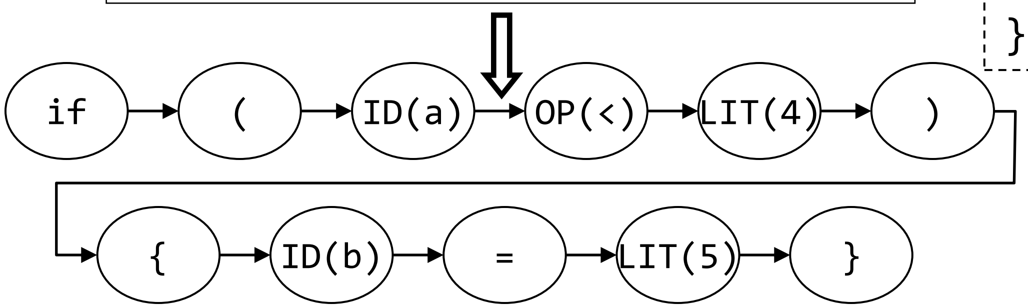
No universally accepted argument

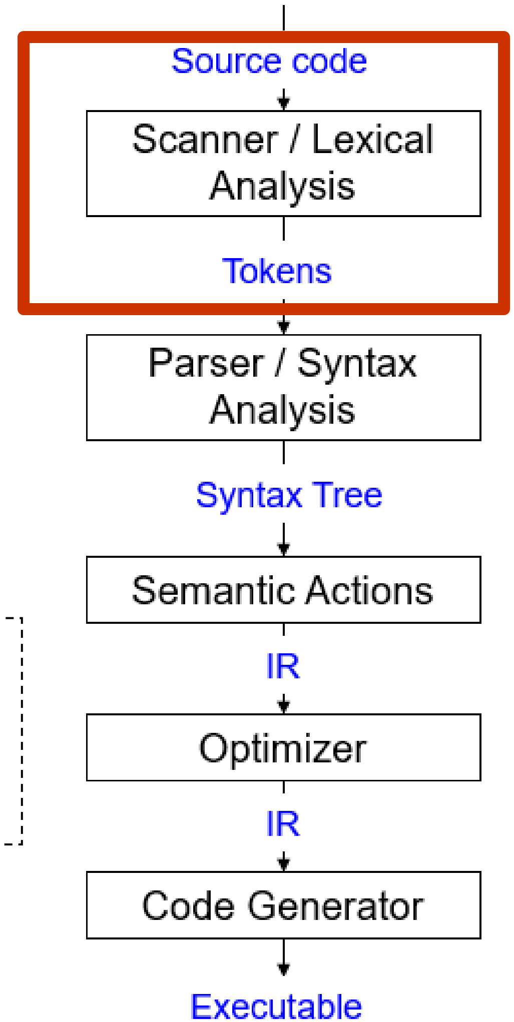
Scanner - Overview

- 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.

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

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





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)

Scanner - Tasks

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

2. Identify the *class* of the substring identified
   - Examples: 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 follow some pattern

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

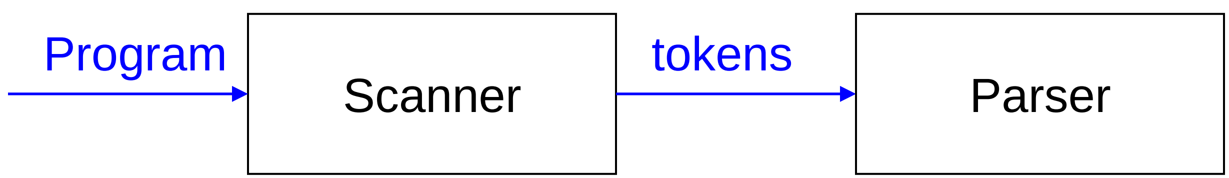
- How many tokens of class *identifier* exist in the code below?

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

Scanner Output

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

A string / lexeme / substring of program text



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;`

Scanners – interesting examples

- 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…

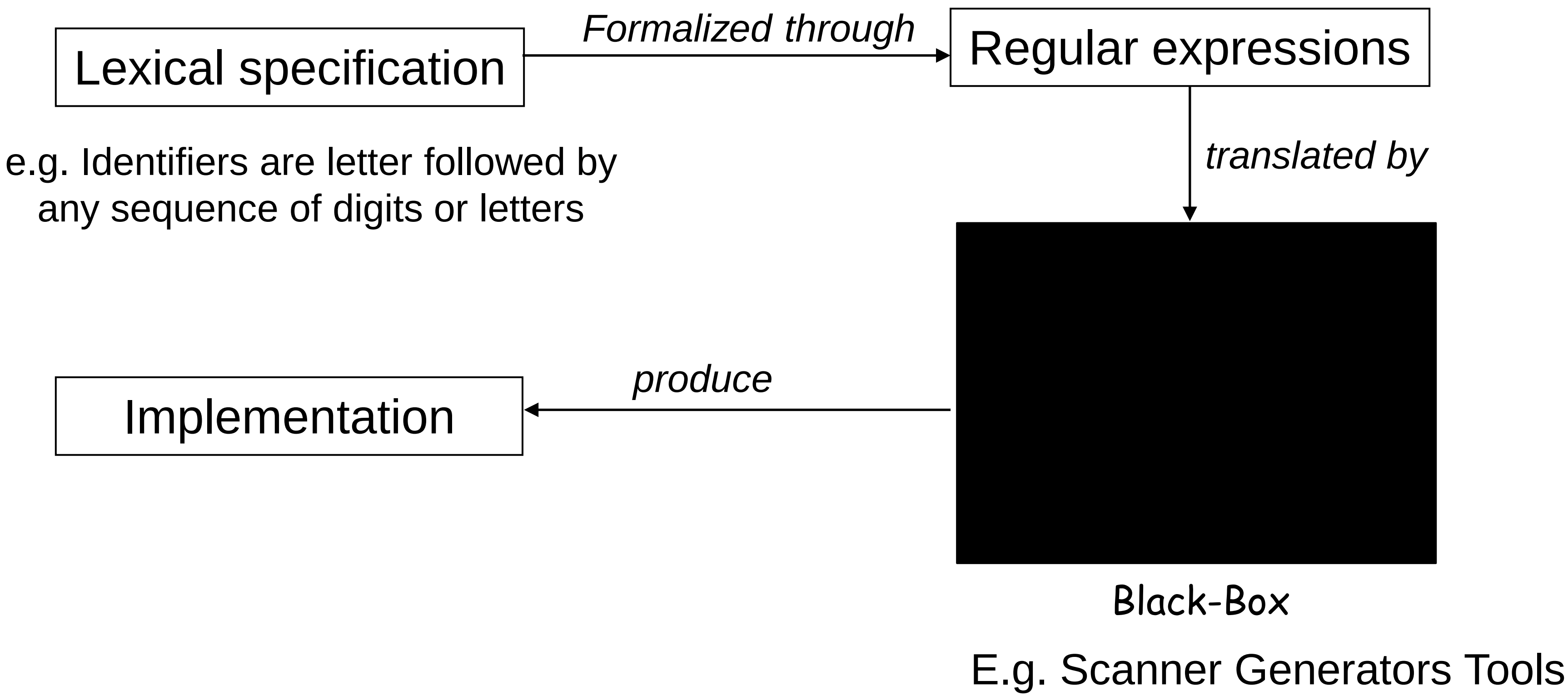
We always need to *look-ahead* to identify tokens

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

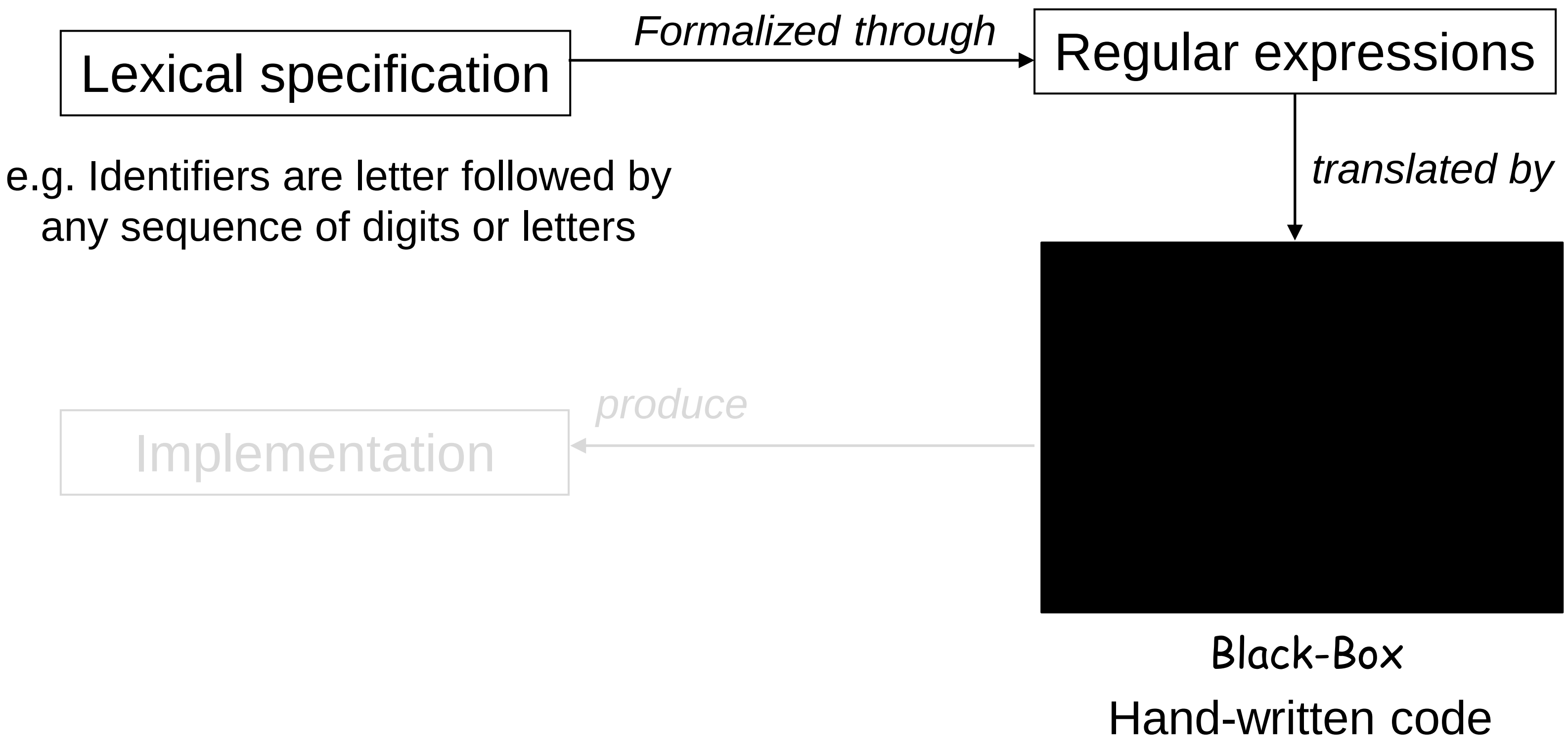
Scanners – what do we need to know?

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



Scanner / Lexical Analyzer - flowchart



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.*

Exercise

- https://forms.gle/crJ2cPYKsx3wywNe6

Regular Expressions

- Used to define the structure of tokens
- Regular sets:

Formal: a language that can be defined by regular expressions

Informal: a set of strings defined by regular expressions

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

Suggested Reading

- Alfred V. Aho, Monica S. Lam, Ravi Sethi and Jeffrey D.Ullman: Compilers: Principles, Techniques, and Tools, 2/E, AddisonWesley 2007
  - Chapter 3 (Sections: 3.1, 3,3, 3.6 to 3.9)
- Fisher and LeBlanc: Crafting a Compiler with C
  - Chapter 3 (Sections 3.1 to 3.4, 3.6, 3.7)