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

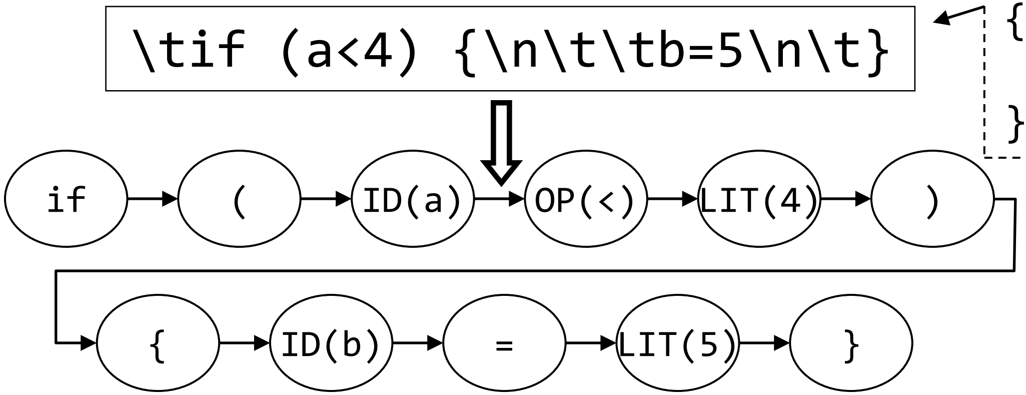
- What is a good Programming Language?

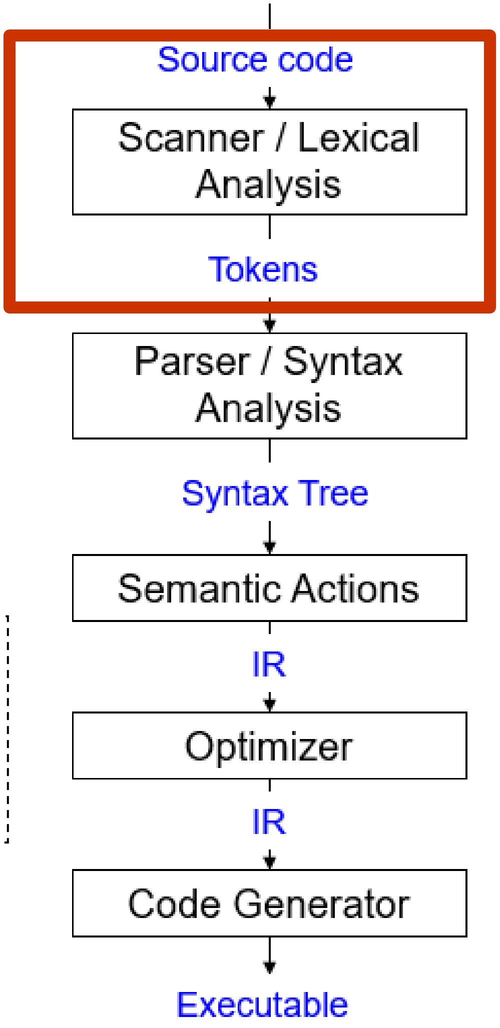
No universally accepted argument

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
}
```





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

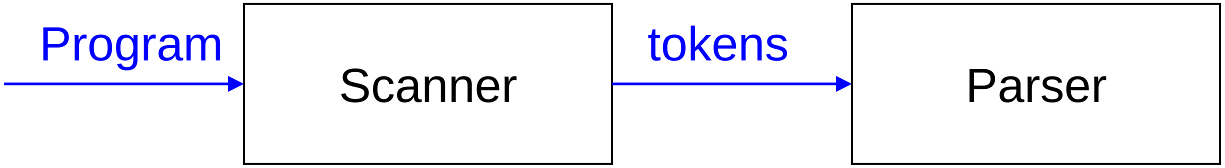
- How many tokens of class *identifier* exist in the code below?

```
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



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;`

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…

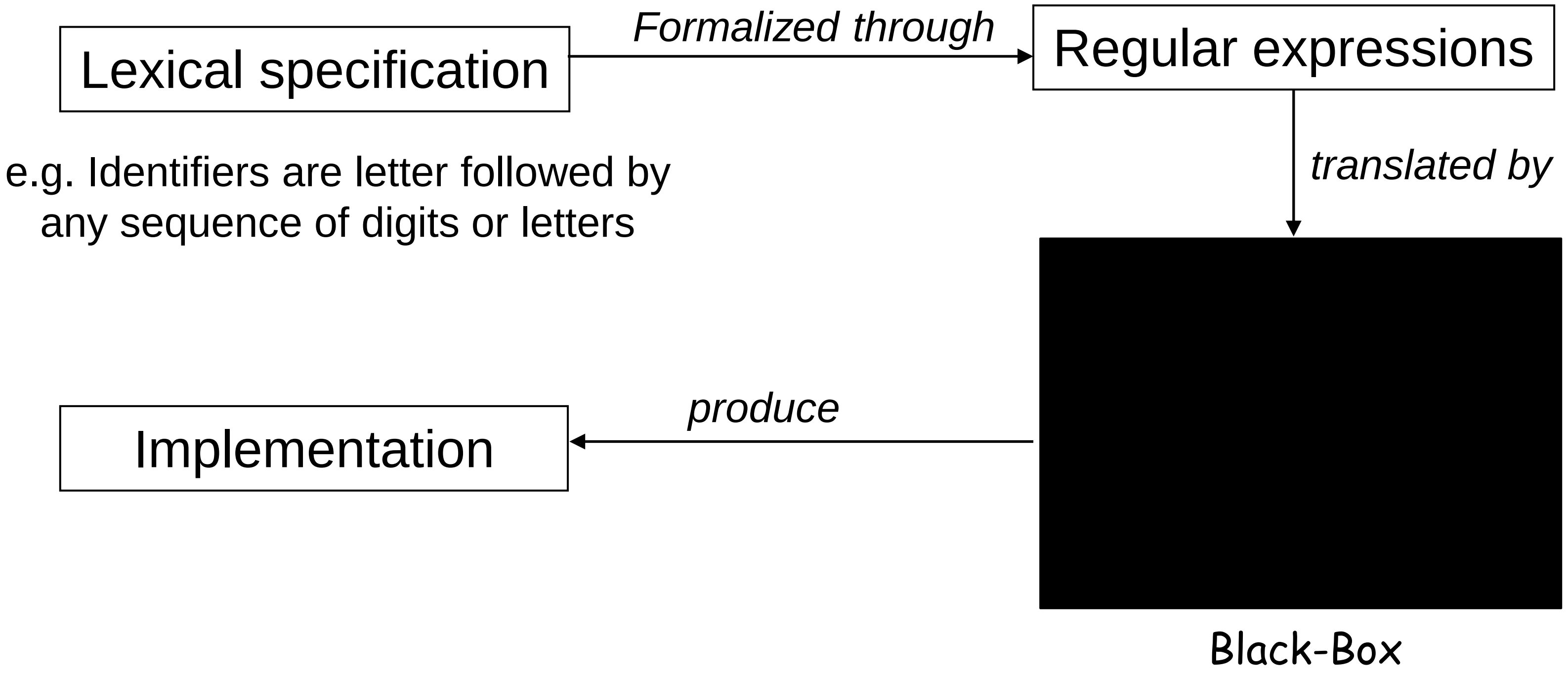
We always need to *look-ahead* to identify tokens

….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



Lexical specification

Formalized through

Regular expressions

translated by

Black-Box

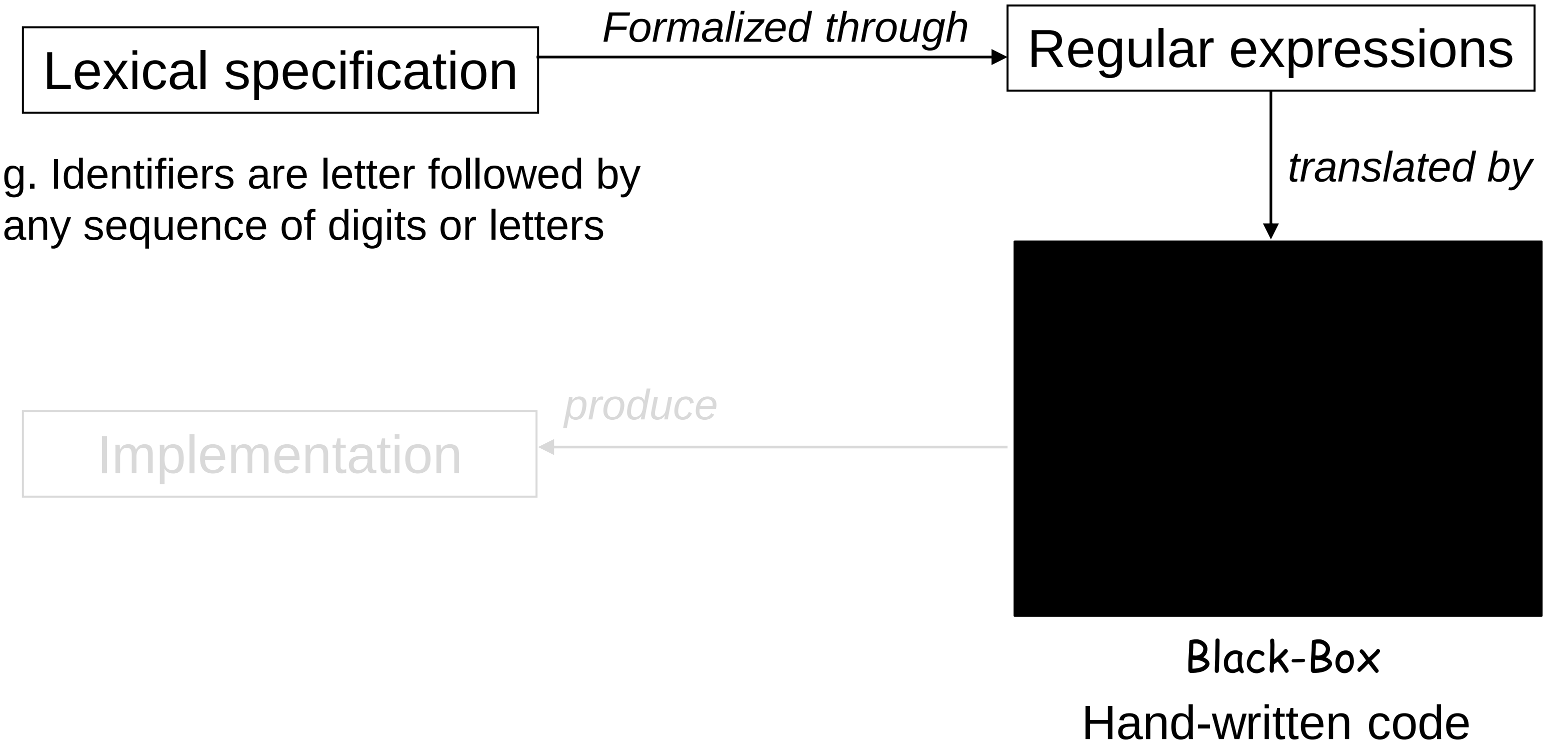
produce

Implementation

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

E.g. Scanner Generators Tools

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.*

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:

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)`

slide courtesy: Milind Kulkarni

Regular Expressions for Lexical Specifications

- Digit: D = (0|1|2|3|4|5|6|7|8|9) OR [0-9]
- Letter: L = [A-Za-z]
- 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(#))* ##

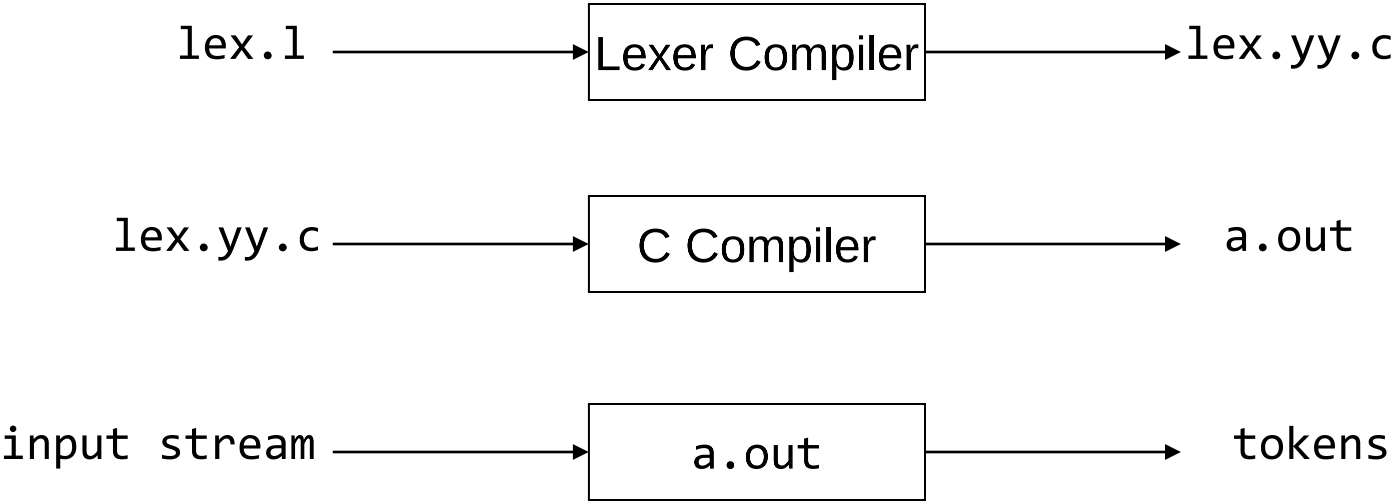
slide courtesy: Milind Kulkarni

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}`

slide courtesy: Milind Kulkarni

Lex (Flex)


lex.l
Lexer Compiler
lex.yy.c
lex.yy.c
C Compiler
a.out
input stream
a.out
tokens

Lex (Flex)

- Format of lex.l

```
Declarations
%%
Translation rules
%%
Auxiliary functions
```

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

slide courtesy: Milind Kulkarni

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

Summary

- We saw what it takes to write a scanner:
  - Specify how to identify token classes (using regexps)
  - Convert the regexps to code that identifies a *prefix* of the input program text as a *lexeme* matching one of the token classes
    - Can use tools for automatic code generation (e.g. `Lex / Flex / ANTLR`)
      - *How do these tools convert regexps to code? Finite Automata*
    - OR write scanner code manually

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)