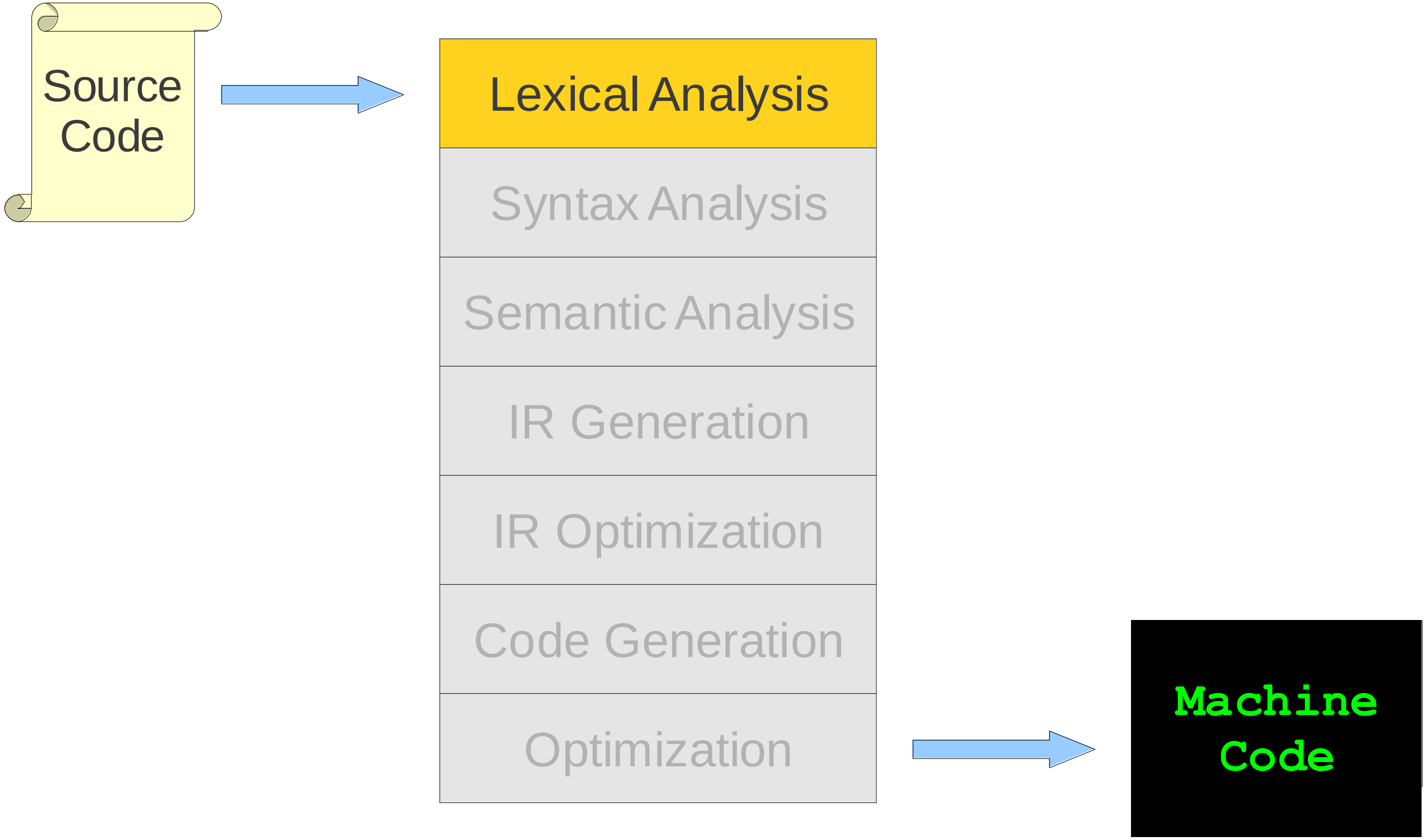
Compiler
- 2-2. Lexical Analysis -

JIEUNG KIM

YONSEI UNIVERSITY

Where are we?


Source Code
Lexical Analysis
Syntax Analysis
Semantic Analysis
IR Generation
IR Optimization
Code Generation
Optimization
Machine Code

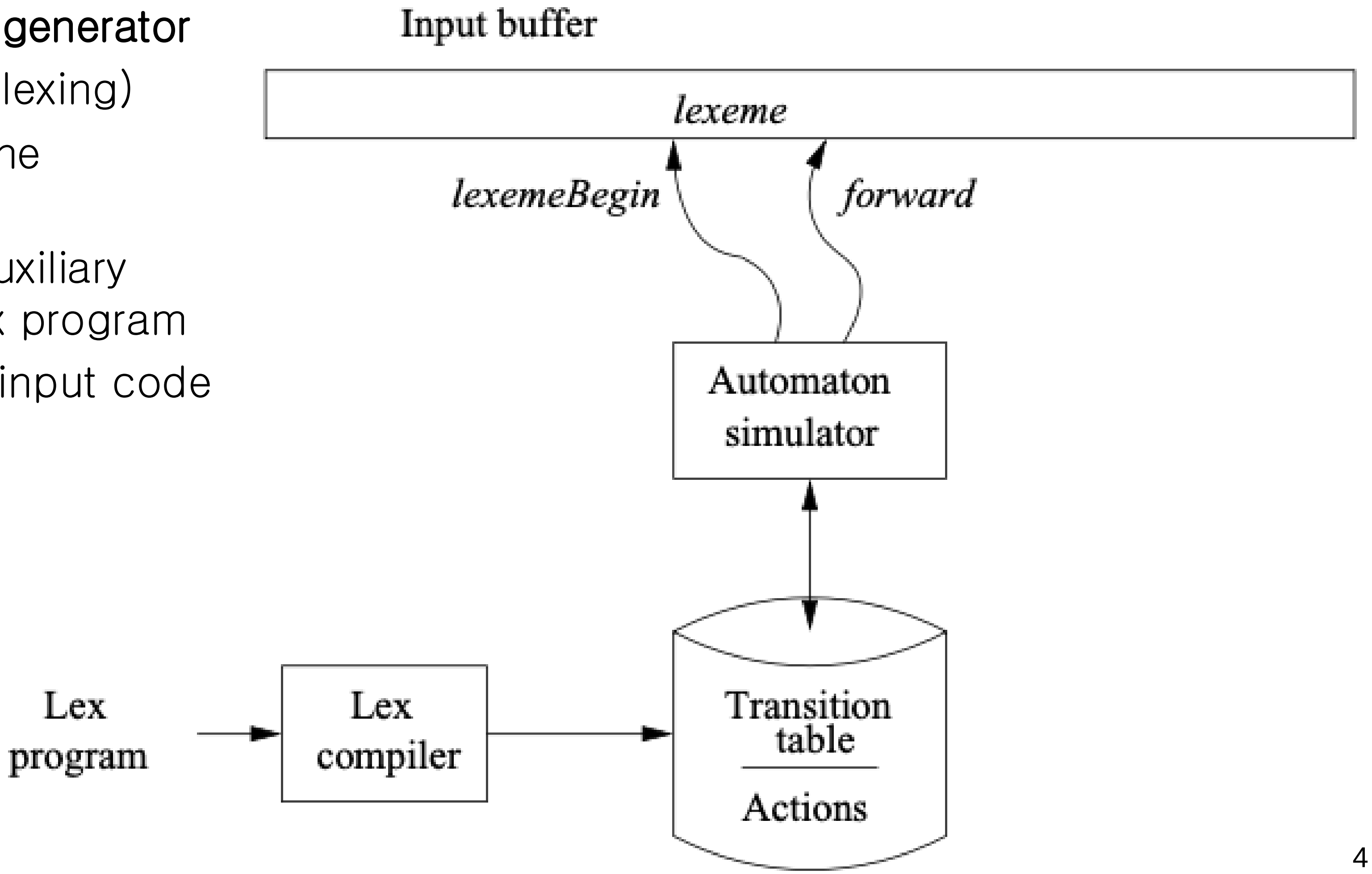
Outlines

- Basic concepts of formal grammars
- Role of the lexical analyzer
- Choose a token
- Finite automata
- Regular expression
- Specification of tokens
- Recognition of tokens
- Challenges in scanning
- Error handling
- Lex : lexical analyzer generator

YONSEI UNIVERSITY

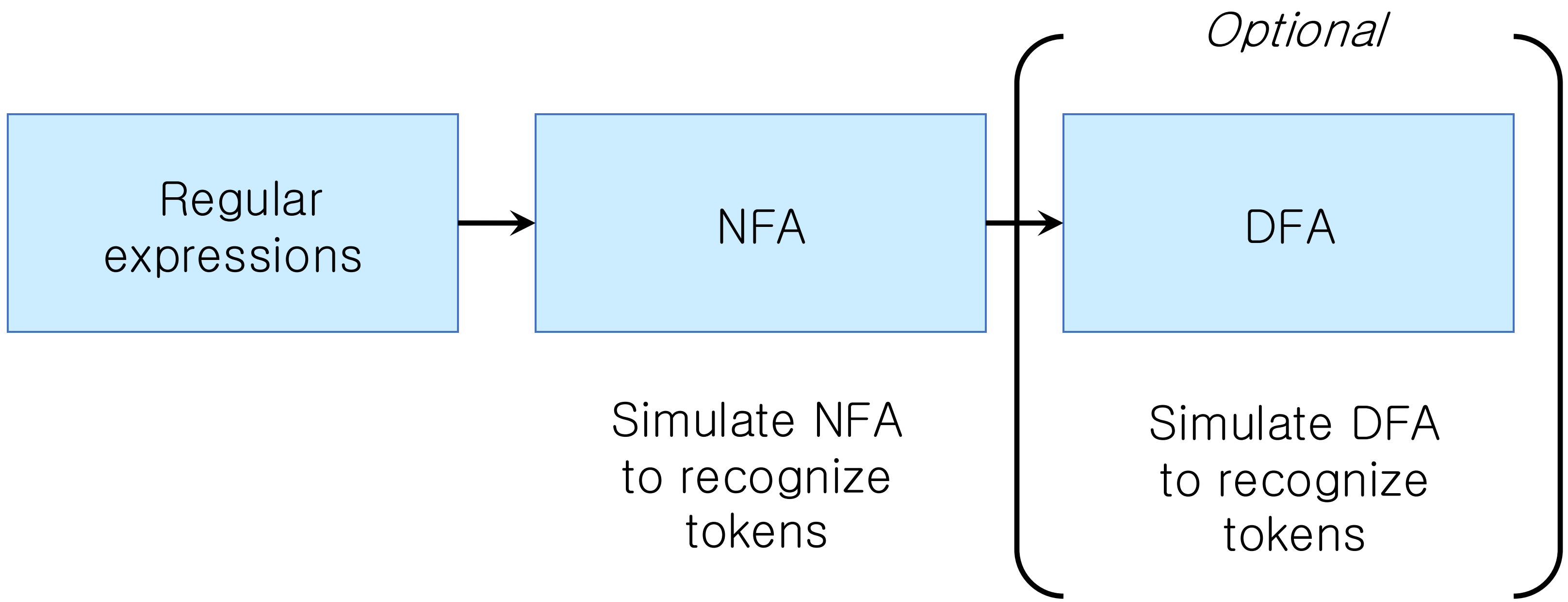
Design of lexical-analyzer generator

- Design of lexical-analyzer generator
  - Automaton simulator (lexing)
  - A transition table for the automaton
    - Directly passed auxiliary functions form lex program
    - The actions from input code



Design of a lexical analyzer generator

- Design of lexical-analyzer generator
  - Translate regular expressions to NFA
  - Translate NFA to an efficient DFA



Specification of tokens

(lexical specification of programming languages)

Specification of tokens

- Specification of tokens
  - Specifying all lexeme patterns is not efficient
  - Use Regular Expressions
- Example

```
num → digits (. Digits)? ( E (+|-)? digits )?
digit → [0-9]
digits → digit+
```

Specification of tokens

Grammars

$$
\begin{aligned}
\textit{stmt} &\rightarrow \textbf{if}\ \textit{expr}\ \textbf{then}\ \textit{stmt} \\
&\quad |\ \textbf{if}\ \textit{expr}\ \textbf{then}\ \textit{stmt}\ \textbf{else}\ \textit{stmt} \\
&\quad |\ \varepsilon \\
\textit{expr} &\rightarrow \textit{term}\ \textbf{relop}\ \textit{term} \\
&\quad |\ \textit{term} \\
\textit{term} &\rightarrow \texttt{id} \\
&\quad |\ \texttt{num}
\end{aligned}
$$

Regular definitions

$$
\begin{aligned}
\textbf{if} &\rightarrow \textbf{if} \\
\textbf{then} &\rightarrow \textbf{then} \\
\textbf{else} &\rightarrow \textbf{else} \\
\textbf{relop} &\rightarrow \texttt{<}\ |\ \texttt{>}\ |\ \texttt{<=}\ |\ \texttt{>=}\ |\ \texttt{=}\ |\ \texttt{<>} \\
\texttt{id} &\rightarrow \texttt{letter (letter | digit)}^{*} \\
\texttt{num} &\rightarrow \texttt{digits (. digis)? (E (+|-)? digits)?} \\
\texttt{letter} &\rightarrow \texttt{[A-Za-z]} \\
\texttt{digit} &\rightarrow \texttt{[0-9]} \\
\texttt{digits} &\rightarrow \texttt{digit}^{+}
\end{aligned}
$$

Recognition of tokens

(using finite automata to recognize regular expressions)

Recognition of tokens

- Recognition of tokens
  - Finite automata can be used to recognize strings generated by regular expressions
  - Can build by hand or automatically
    - Not totally straightforward, but can be done systematically
    - Tools like Lex, Flex, Jlex, ANTLR do this automatically, given a set of REs for tokens

Recognition of tokens

- Recognition of tokens

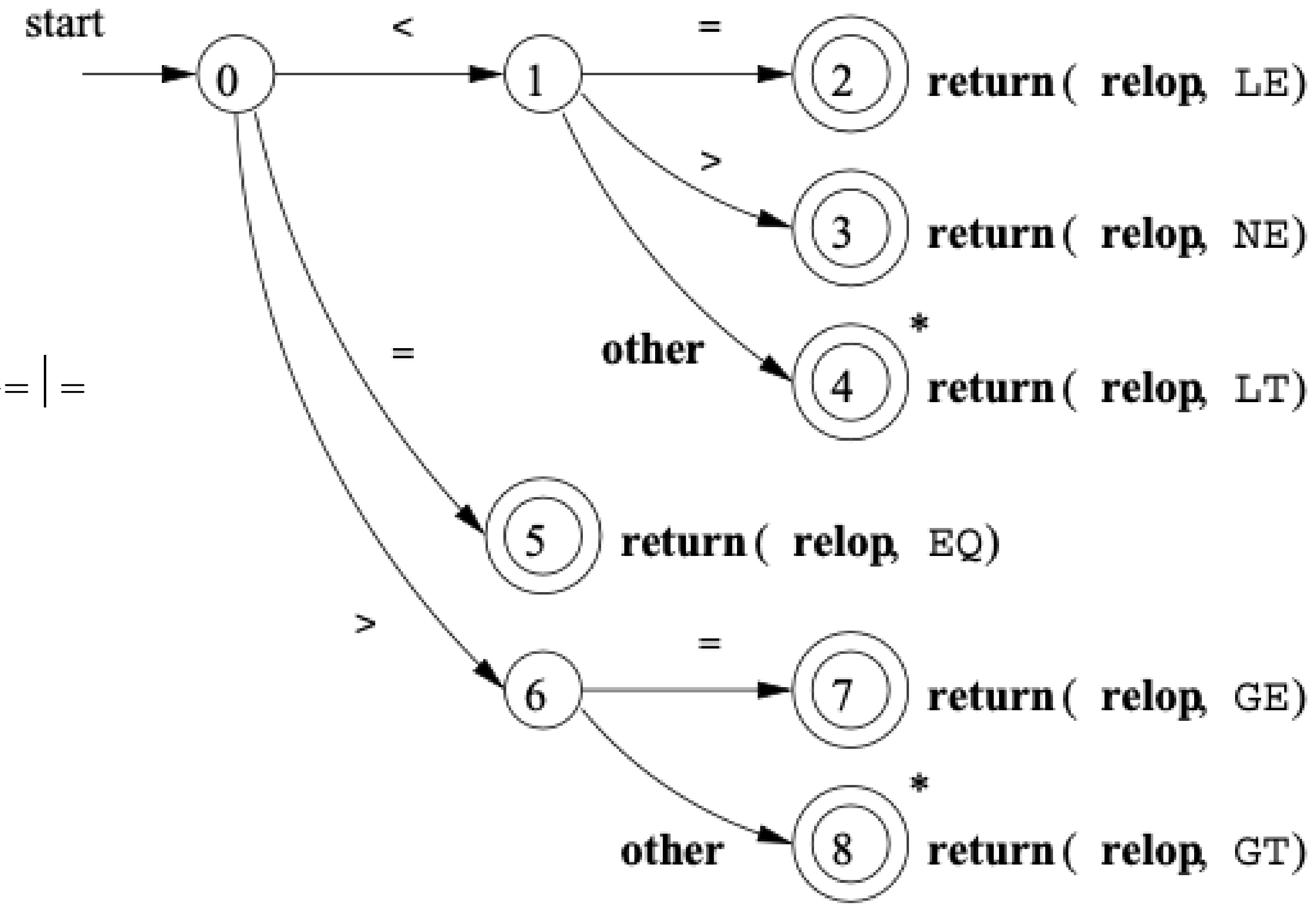
```
     if → if
   then → then
   else → else
  relop → < | > | <= | >= | = | <>
     id → letter ( letter | digit )*
    num → digits (. digis)? ( E (+|-)? digits )?
 letter → [A-Za-z]
  digit → [0-9]
 digits → digit+
```

The issues are "*how to recognize the keywords– if, then else, relop, id, number*"

Recognition of tokens

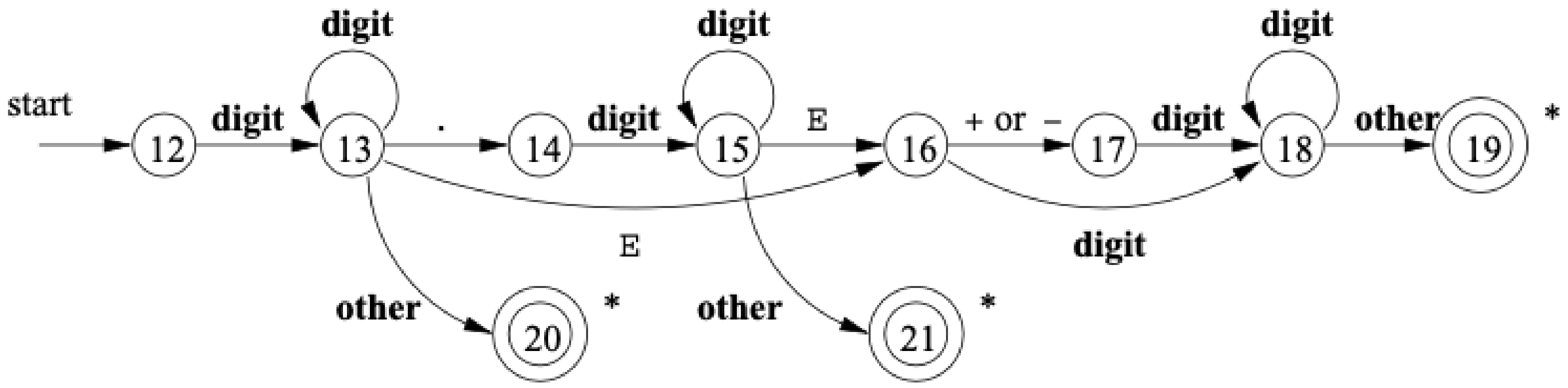
- Transition diagram for relop

relop $\rightarrow$ < | <= | <> | > | >= | =



Recognition of tokens

- Transition diagram for unsigned numbers

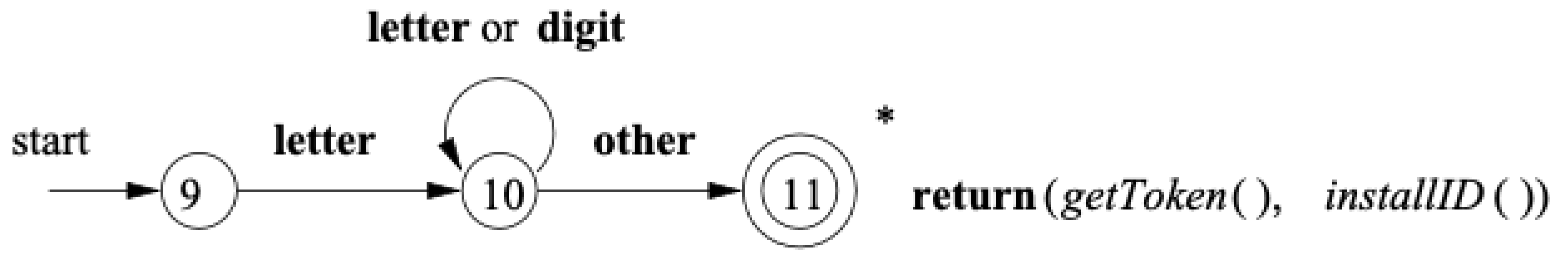


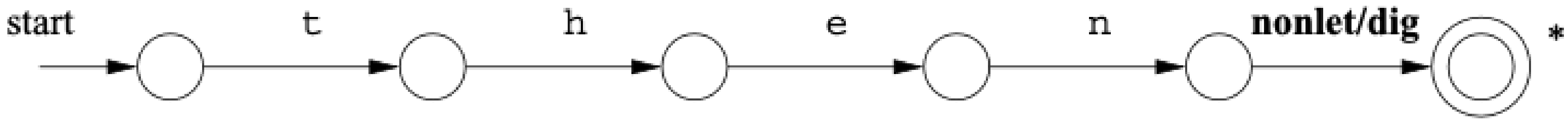
Recognition of tokens

- **Transition diagram for reserved keywords and IDs**
  - Two ways to handle reserved word
  - Install keyword symbol table initially
    - Hand-written scanner: look up identifier-like things in table of keywords to classify (good application of perfect hashing)
  - Create separate diagram for each keyword
    - Lots of states, but efficient (no extra lookup step)
    - Machine-generated scanner: generate DFA will appropriate transitions to recognize keywords

Recognition of tokens

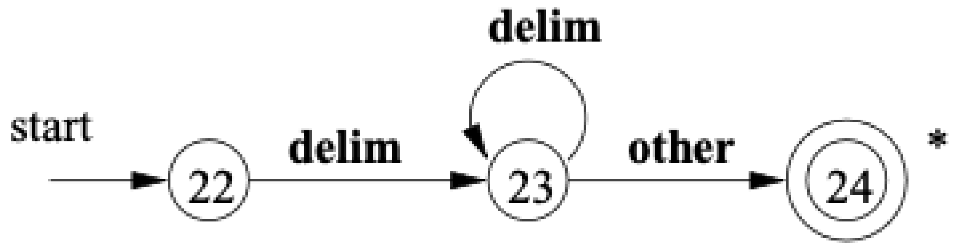
- Transition diagram for reserved words and IDs





Recognition of tokens

- Transition diagram for white spaces



Simple hand-written scanner example

- **Example: DFA and hand-written scanner (ad hoc scanners)**
  - Idea: show a hand-written DFA for some typical programming language constructs
    - Then use to construct hand-written scanner
  - Setting: scanner is called whenever the parser needs a new token
    - Scanner stores current position in input
    - Starting there, use a DFA to recognize the longest possible input sequence that makes up a token and return that token

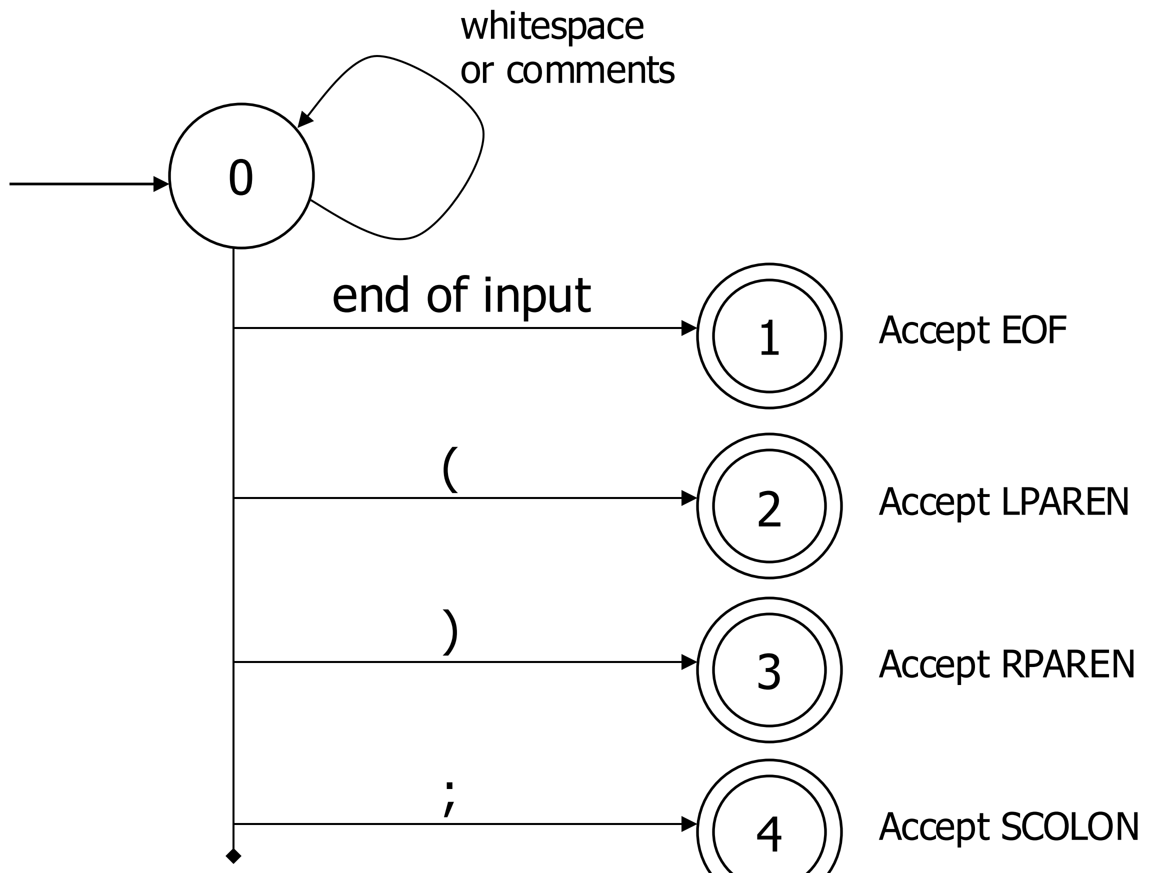
YONSEI UNIVERSITY

Simple hand-written scanner example

- **Example: DFA for hand-written scanner (ad hoc scanners)**
  - Loop and switch scanner
  - Big nested switch/case statements
  - Lots of `gets()/ungetc()` calls
  - Must be error-prone, use only if
    - The lexical structure of the language is very simple
    - The tools do not provide what you need for your token definition
    - Changing keyword is problematic
  - Very difficult to show correctness

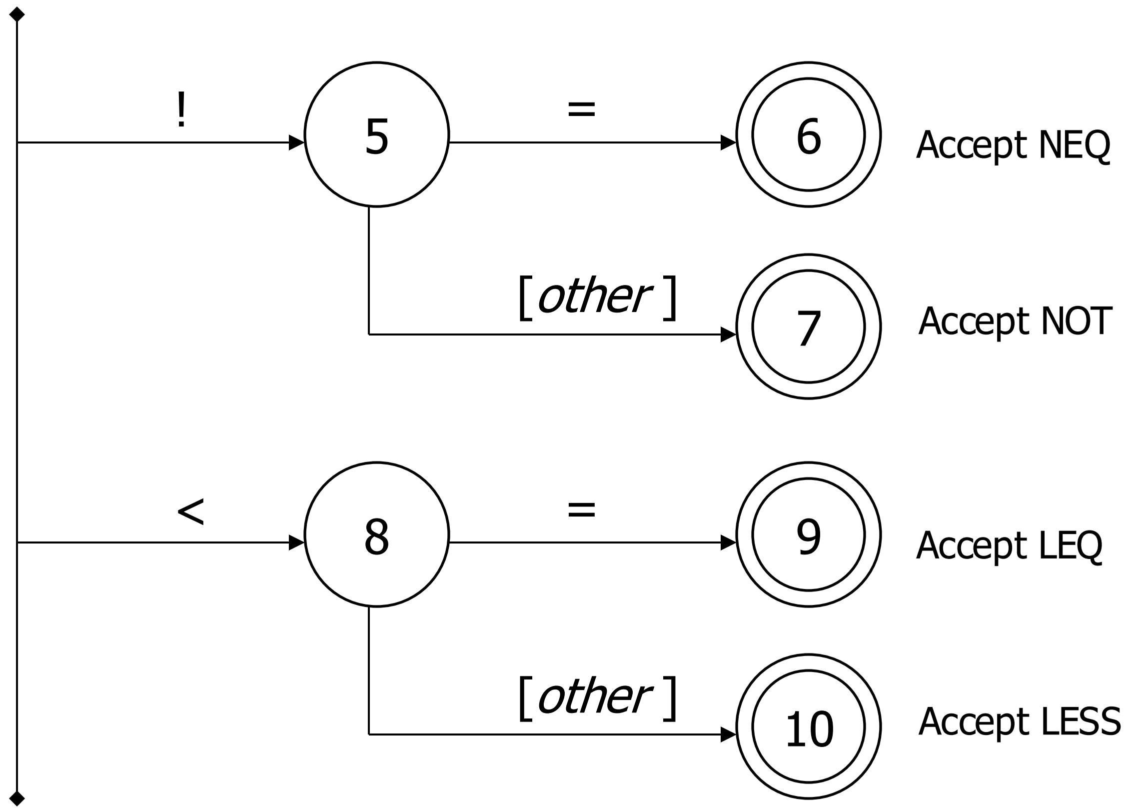
Simple hand-written scanner example

- Example 2 (1)



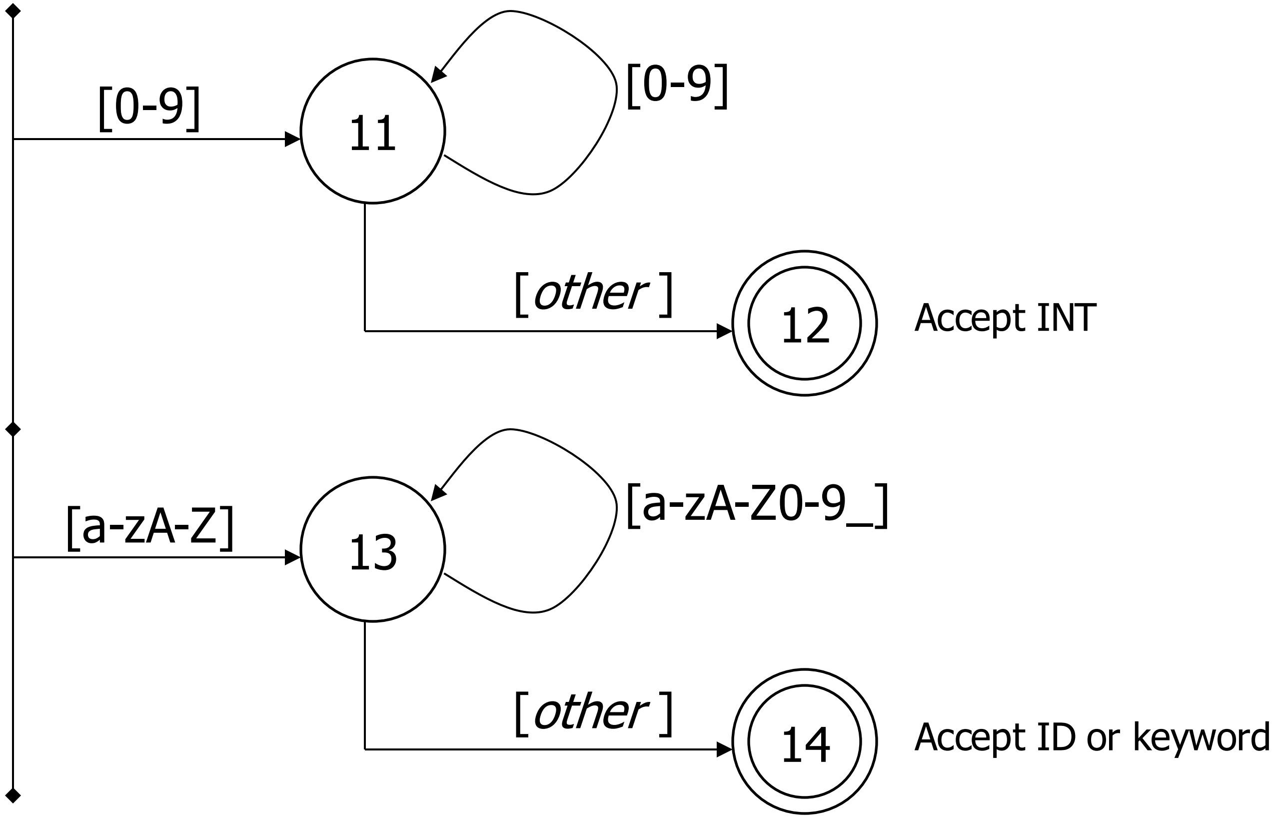
Simple hand-written scanner example

- Example 2 (3)



Simple hand-written scanner example

- Example 2 (4)



Simple hand-written scanner example

- Implementing a scanner by hand – token representation

```
public class Token {
    public int kind;                      // token's lexical class
    public int intVal;                    // integer value if class = INT
    public String id;                     // actual identifier if class = ID
    // lexical classes
    public static final int EOF = 0;      // "end of file" token
    public static final int ID   = 1;     // identifier, not keyword
    public static final int INT = 2;      // integer
    public static final int LPAREN = 4;
    public static final int SCOLN   = 5;
    public static final int WHILE   = 6;
    // etc. etc. etc. …
```

Simple hand-written scanner example

- Implementing a scanner by hand – read inputs

```
// global state and methods

static char nextch;    // next unprocessed input character

// advance to next input char
void getch() { … }

// skip whitespace and comments
void skipWhitespace() { … }
```

Simple hand-written scanner example

- Implementing a scanner by hand – scanner getToken() method

```
// return next input token
public Token getToken() {
  Token result;

  skipWhiteSpace();

  if (no more input) {
    result = new Token(Token.EOF);
    return result;
  }

  switch(nextch) {
    case '(': result = new Token(Token.LPAREN); getch(); return result;
    case ‘)': result = new Token(Token.RPAREN); getch(); return result;
    case ‘;': result = new Token(Token.SCOLON); getch(); return result;

    // etc. …
```

YONSEI UNIVERSITY

Simple hand-written scanner example

- Implementing a scanner by hand – scanner getToken() method

```
case '!’:                                        // ! or !=
  getch();
  if (nextch == '=') {
    result = new Token(Token.NEQ); getch(); return result;
  } else {
    result = new Token(Token.NOT); return result;
  }

case '<‘:                                        // < or <=
  getch();
    if (nextch == '=‘) {
      result = new Token(Token.LEQ); getch(); return result;
    } else {
      result = new Token(Token.LESS); return result;
    }
// etc. …
```

YONSEI UNIVERSITY

Simple hand-written scanner example

- Implementing a scanner by hand – scanner getToken() method

```
case '0': case '1': case '2': case '3': case '4':
case '5': case '6': case '7': case '8': case '9':
  // integer constant
  String num = nextch;
  getch();
  while (nextch is a digit) {
    num = num + nextch; getch();
  }
  result = new Token(Token.INT, Integer(num).intValue());
  return result;
…
```

Simple hand-written scanner example

- Implementing a scanner by hand – scanner getToken() method

```
case 'a': … case 'z':
case 'A': … case ‘Z’:
  // id or keyword
  string s = nextch; getch();
  while (nextch is a letter, digit, or underscore) {
    s = s + nextch; getch();
  }
  if (s is a keyword) {
    result = new Token(keywordTable.getKind(s));
  } else {
    result = new Token(Token.ID, s);
  }
  return result;
```

Challenges in scanning

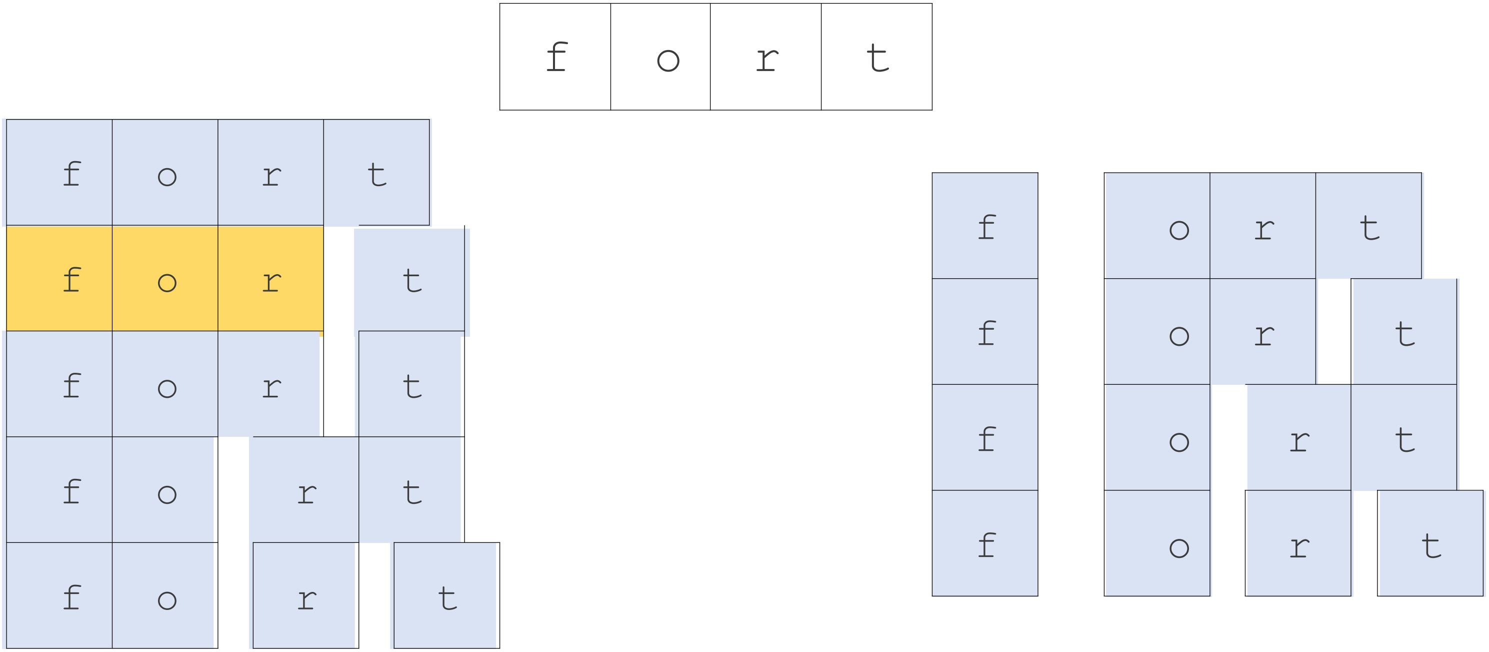
Challenges in scanning

- Challenges
  - How do we determine which lesemes are associated with each token?
  - When there are multiple ways we could scan the input, how do we know which one to pick?
  - How do we address these concerns efficiently?

Challenges in scanning

- Lexing ambiguity

T_For	for
T_ID	[A-Za-z_][A-Za-z0-9_]*



Challenges in scanning

- **Conflict resolution**
  - How do we determine which lesemes are associated with each token?
  - When there are multiple ways we could scan the input, how do we know which one to pick?
  - How do we address these concerns efficiently?
  - Assume all tokens are specified as regular expressions.
  - Algorithm: **Left-to-right scan.**
  - Tiebreaking rule one: **Maximal munch.**
    - Always match the longest possible prefix of the remaining text.

Challenges in scanning

- Lexing ambiguity

T_For	for
T_ID	[A-Za-z_][A-Za-z0-9_]*

f	o	r	t

f	o	r	t

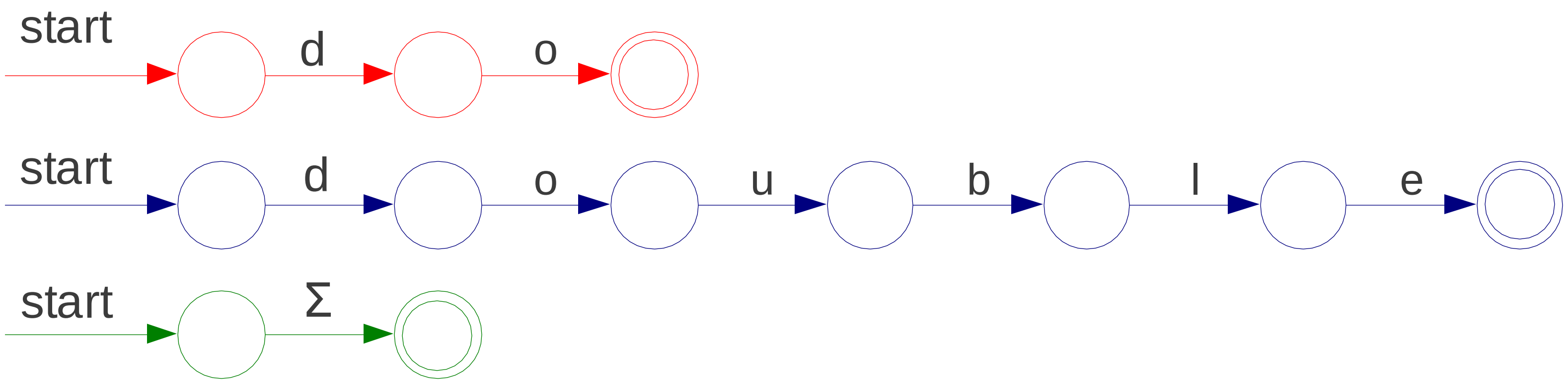
Challenges in scanning

- **Implementing maximal munch**
  - Given a set of regular expressions, how can we use them to implement maximum munch?
  - Idea:
    - Convert expressions to NFAs.
    - Run all NFAs in parallel, keeping track of the last match.
    - When all automata get stuck, report the last match and restart the search at that point.

Challenges in scanning

- Implementing maximal munch

T_DO	do
T_DDOUBLE	double
T_OTHERS	[A-Za-z]

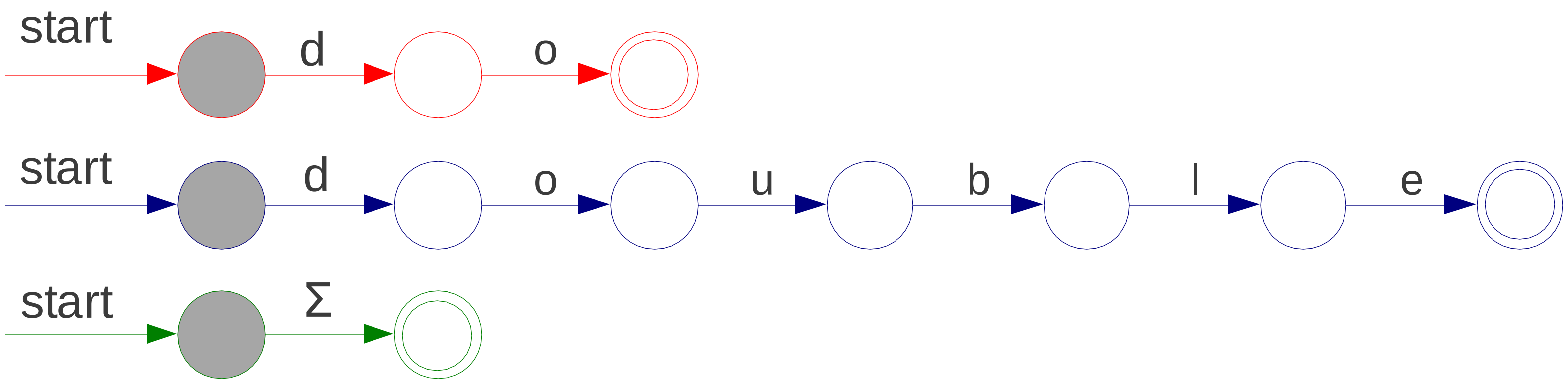


D	O	U	B	D	O	U	B	L	E

Challenges in scanning

- Implementing maximal munch

T_DO	do
T_DDOUBLE	double
T_OTHERS	[A-Za-z]

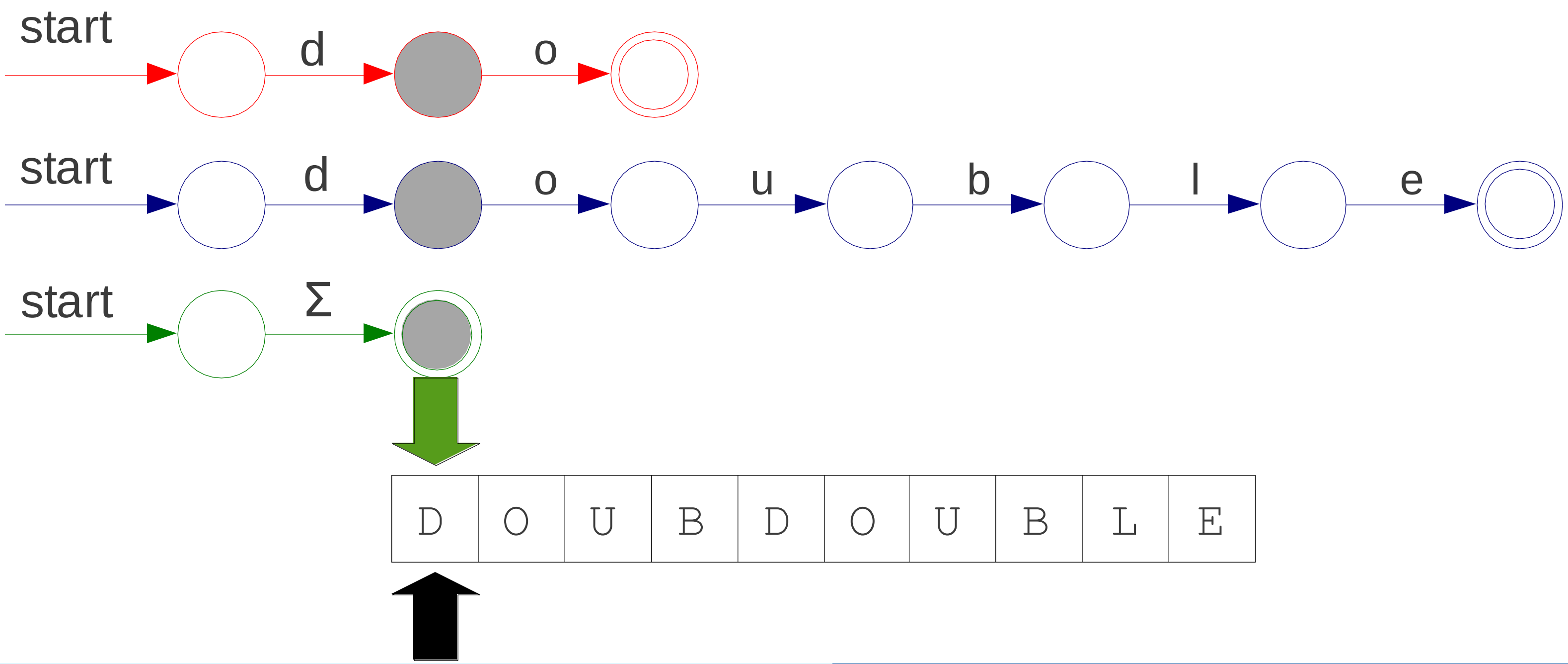


D	O	U	B	D	O	U	B	L	E

Challenges in scanning

- Implementing maximal munch

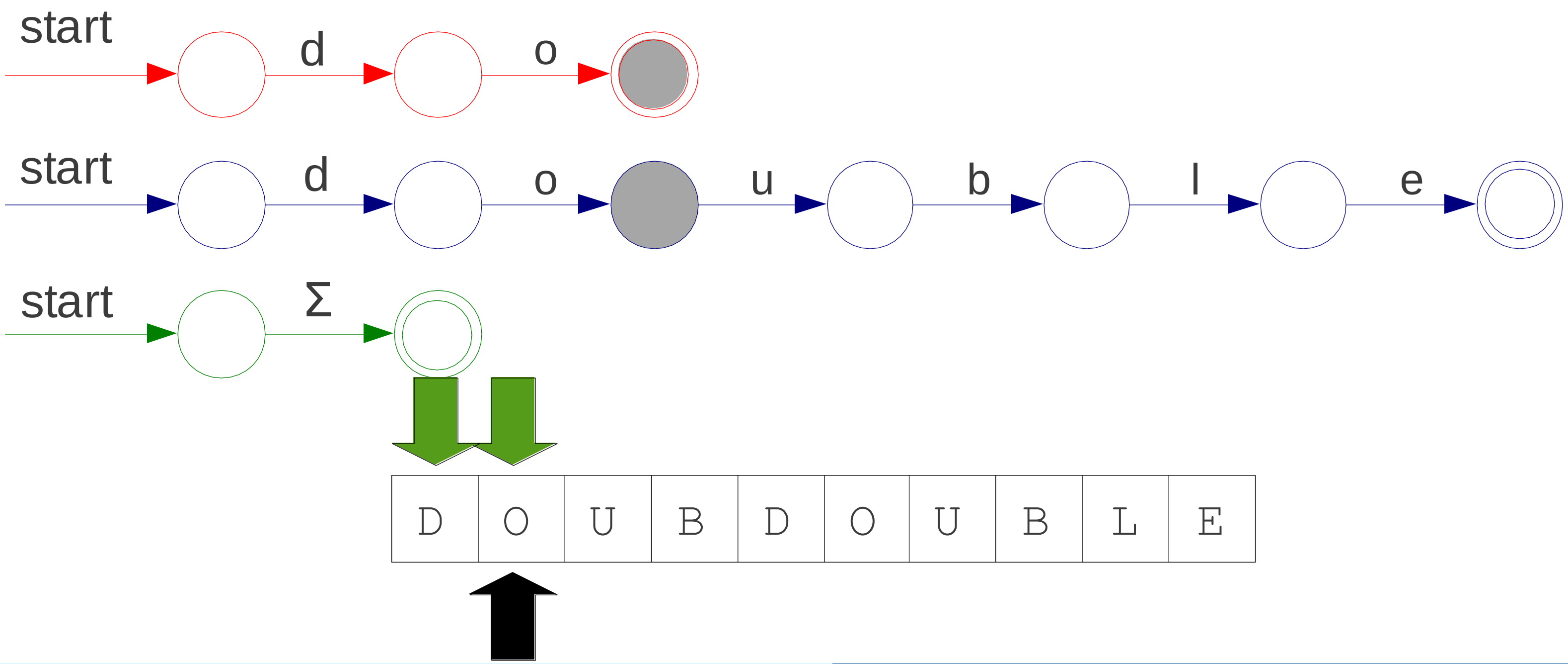
T_DO	do
T_DDOUBLE	double
T_OTHERS	[A-Za-z]



Challenges in scanning

- Implementing maximal munch

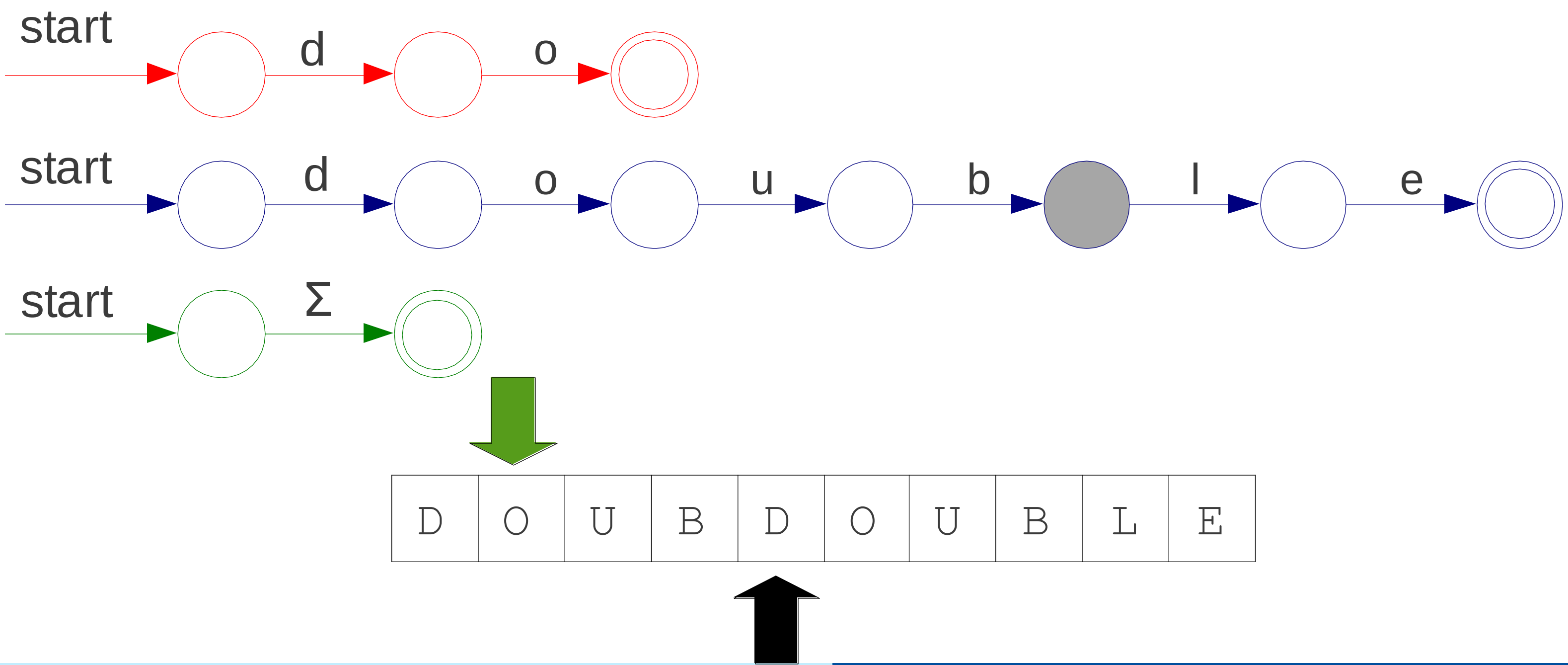
T_DO	do
T_DDOUBLE	double
T_OTHERS	[A-Za-z]



Challenges in scanning

- Implementing maximal munch

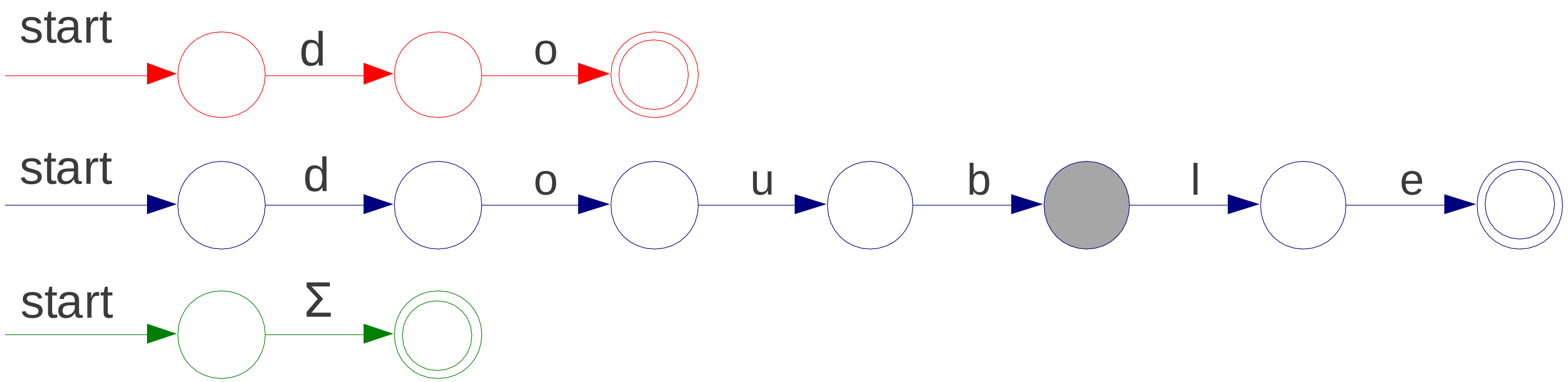
T_DO	do
T_DDOUBLE	double
T_OTHERS	[A-Za-z]

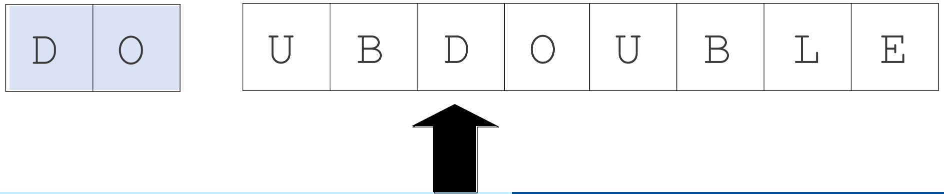


Challenges in scanning

- Implementing maximal munch

T_DO	do
T_DDOUBLE	double
T_OTHERS	[A-Za-z]

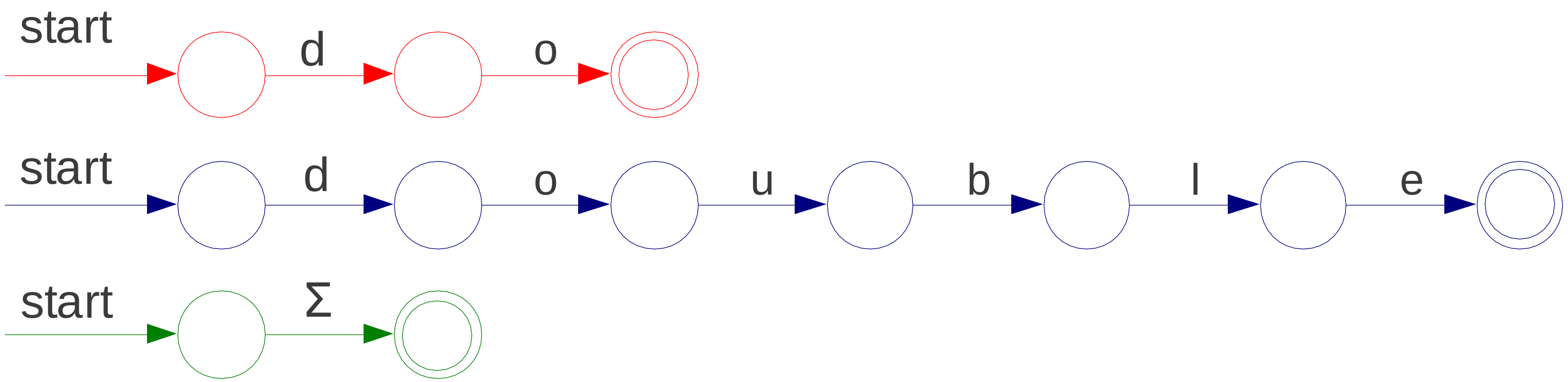


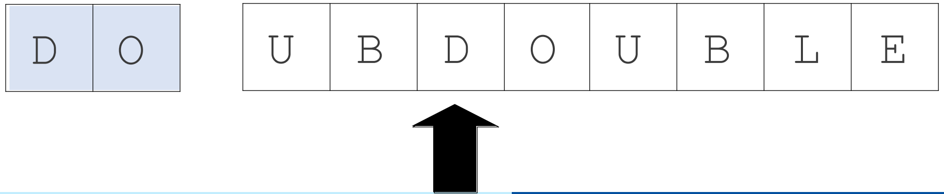


Challenges in scanning

- Implementing maximal munch

T_DO	do
T_DDOUBLE	double
T_OTHERS	[A-Za-z]

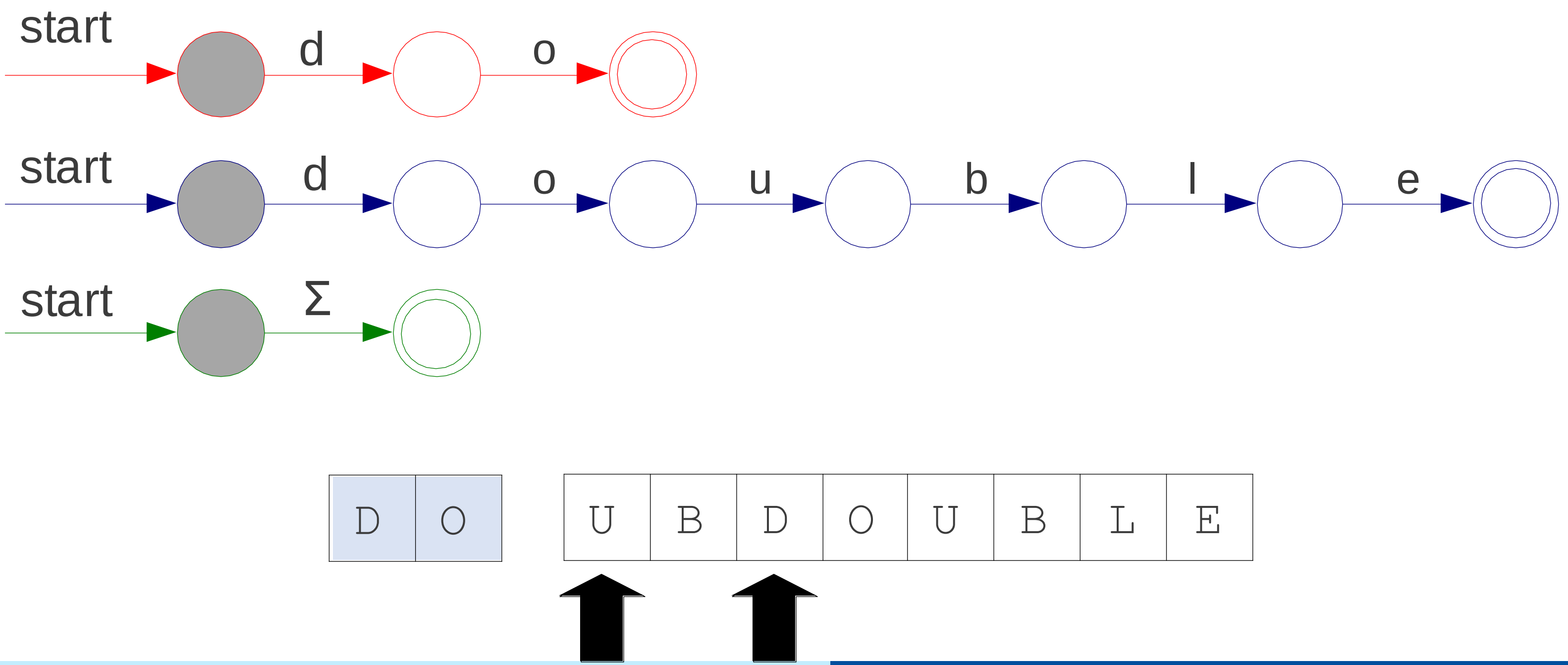




Challenges in scanning

- Implementing maximal munch

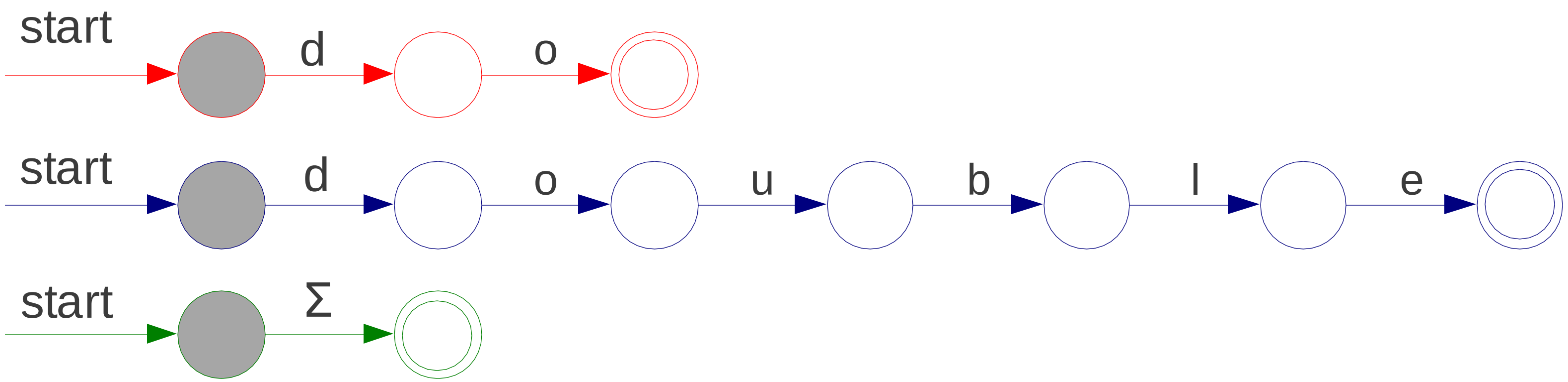
T_DO	do
T_DDOUBLE	double
T_OTHERS	[A-Za-z]



Challenges in scanning

- Implementing maximal munch

T_DO	do
T_DDOUBLE	double
T_OTHERS	[A-Za-z]



D	O		U		B		D	O	U	B	L	E

Challenges in scanning

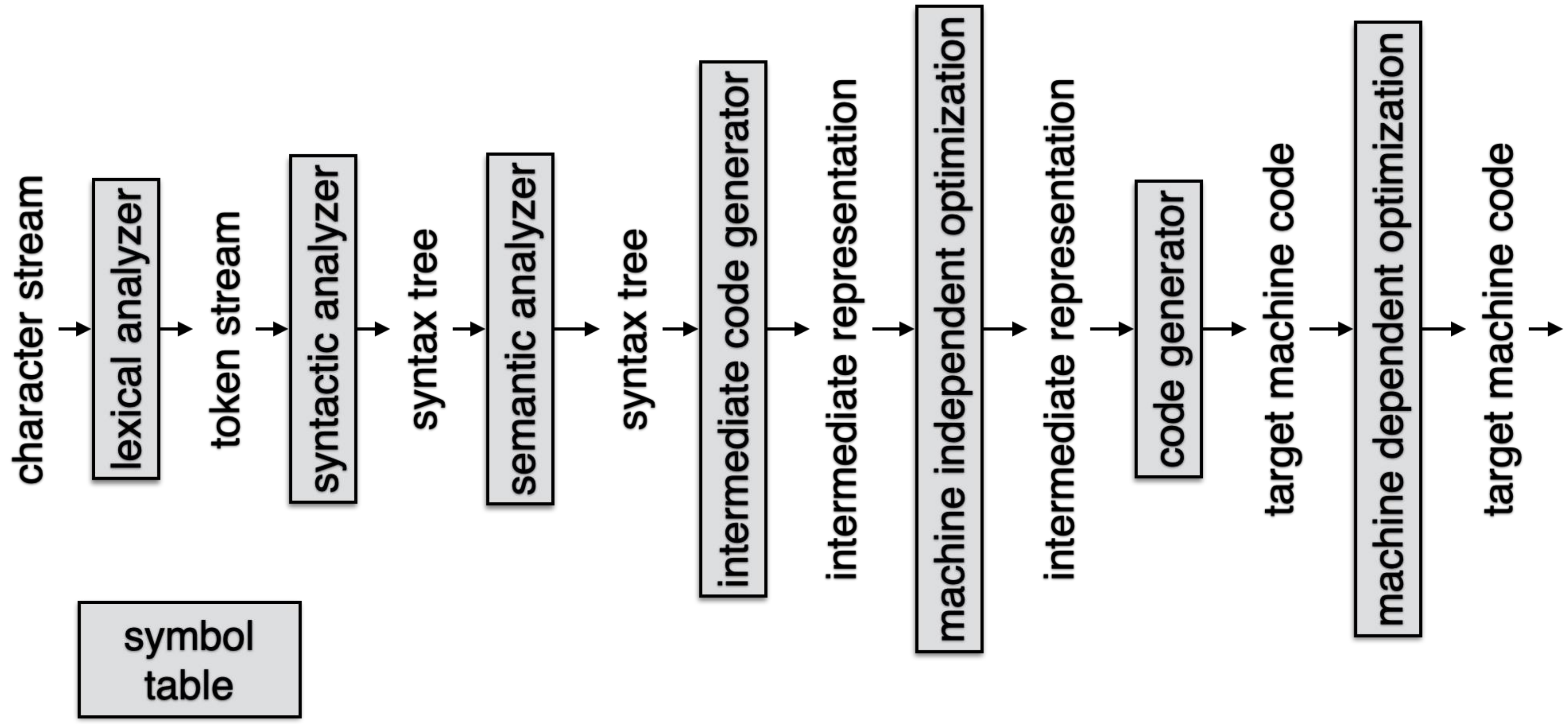
- Other conflicts
  - When two regular expressions apply, choose the one with the greater “priority.”
  - Simple priority system: **Pick the rule that was defined first.**

```
T_DO                 do
T_DDOUBLE            double
T_ID                 [A-Za-z_][A-Za-z0-9_]*
```

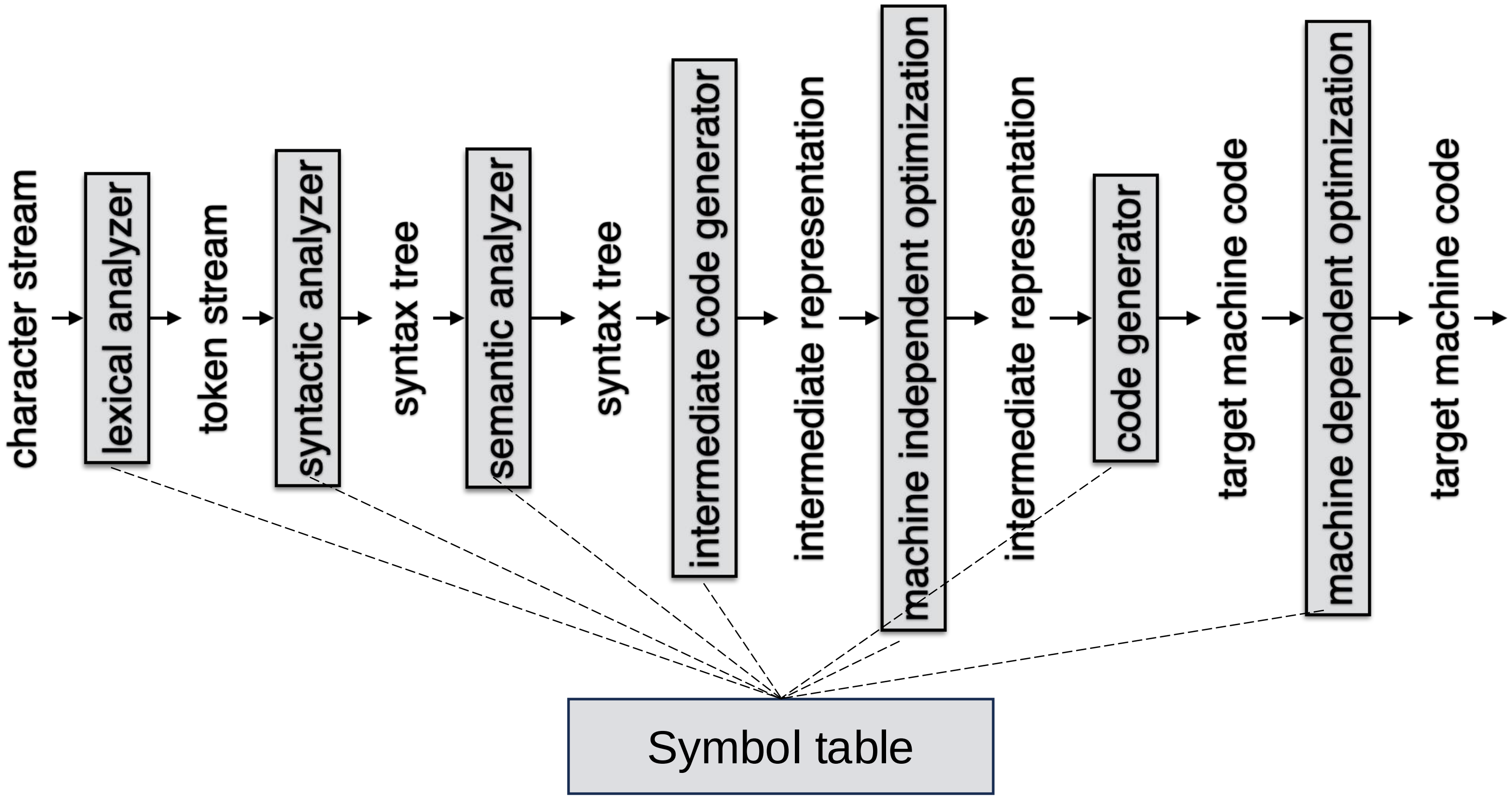
D	O	U	B	L	E

Error handling

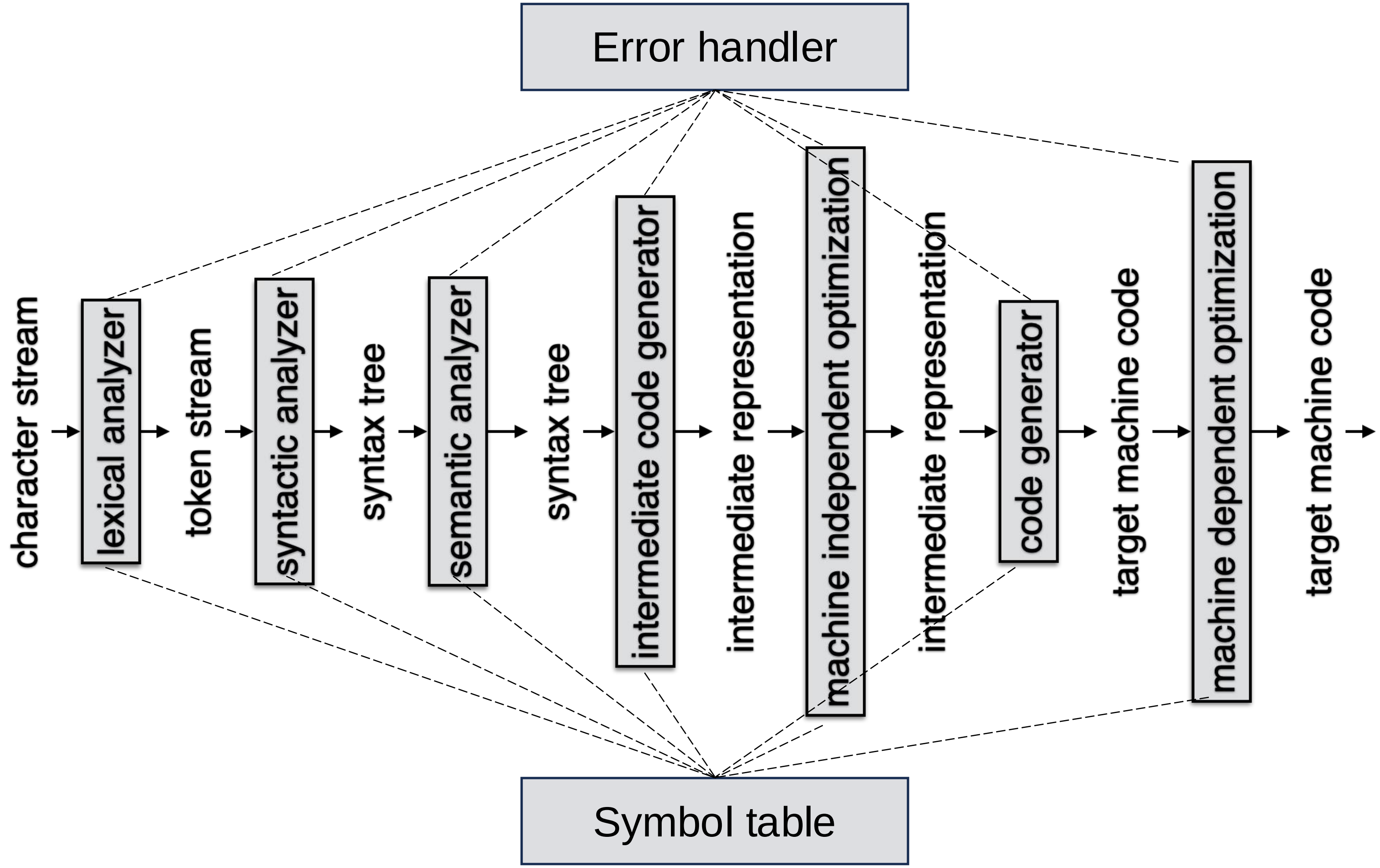
Compiler architecture



Compiler architecture



Compiler architecture



YONSEI UNIVERSITY

Kinds of errors

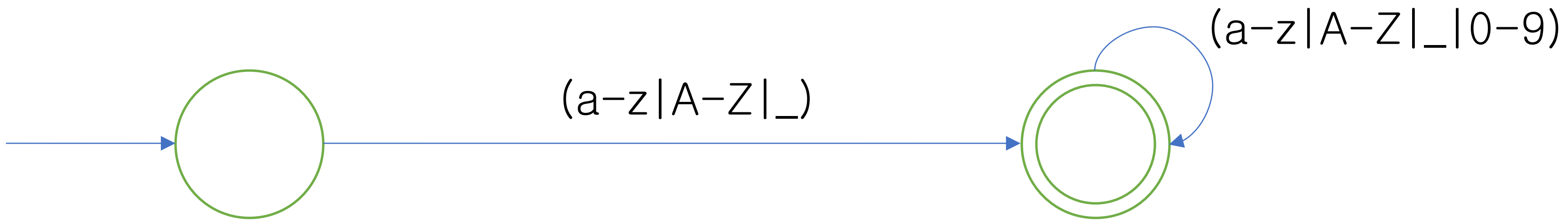
- Errors
  - Compile Time
    - Lexical errors
    - Syntax errors
    - Semantic errors
  - Runtime

YONSEI UNIVERSITY

Lexical errors

- Identifiers that are way too long

Identifier:



C: 31/247 C++: 2024 Python: 79

- Exceeding length of numeric constants

unsinged int i = 4294967300;

Size: 4 bytes 0 ~ 4,294,967,295

Lexical errors

- Numeric constants which are ill-formed

```
int i = 4567$91;
```

- Illegal characters that are absent from the source code.

```
char x[] = “YONSEI UNIVERSITY”;$
```

Lexical error recovery

- Panic-mode recovery
  - Successive characters from the input are removed one at a time until a designated set of synchronizing tokens is found.
  - Synchronizing tokens are delimiters such as; or }
  - The advantage is that it is easy to implement and guarantees not to go into an infinite loop
  - The disadvantage is that a considerable amount of input is skipped without checking it for additional errors

```
int i = 4YONSEI;
```

```
while (condition)
{
  __________
  ----------
  __________
}
```

YONSEI UNIVERSITY

Lexical error recovery

- Transpose of two adjacent characters.

```
unoin test
{
    int x;
    float y;
} T1;
```
→
```
union test
{
    int x;
    float y;
} T1;
```

- Insert a missing character.

`it YONSEI;` → `int YONSEI;`

- Delete an unknown or extra character.

`intt YONSEI;` → `int YONSEI;`

- Replace a character with another.

`itt YONSEI;` → `int YONSEI;`

YONSEI UNIVERSITY

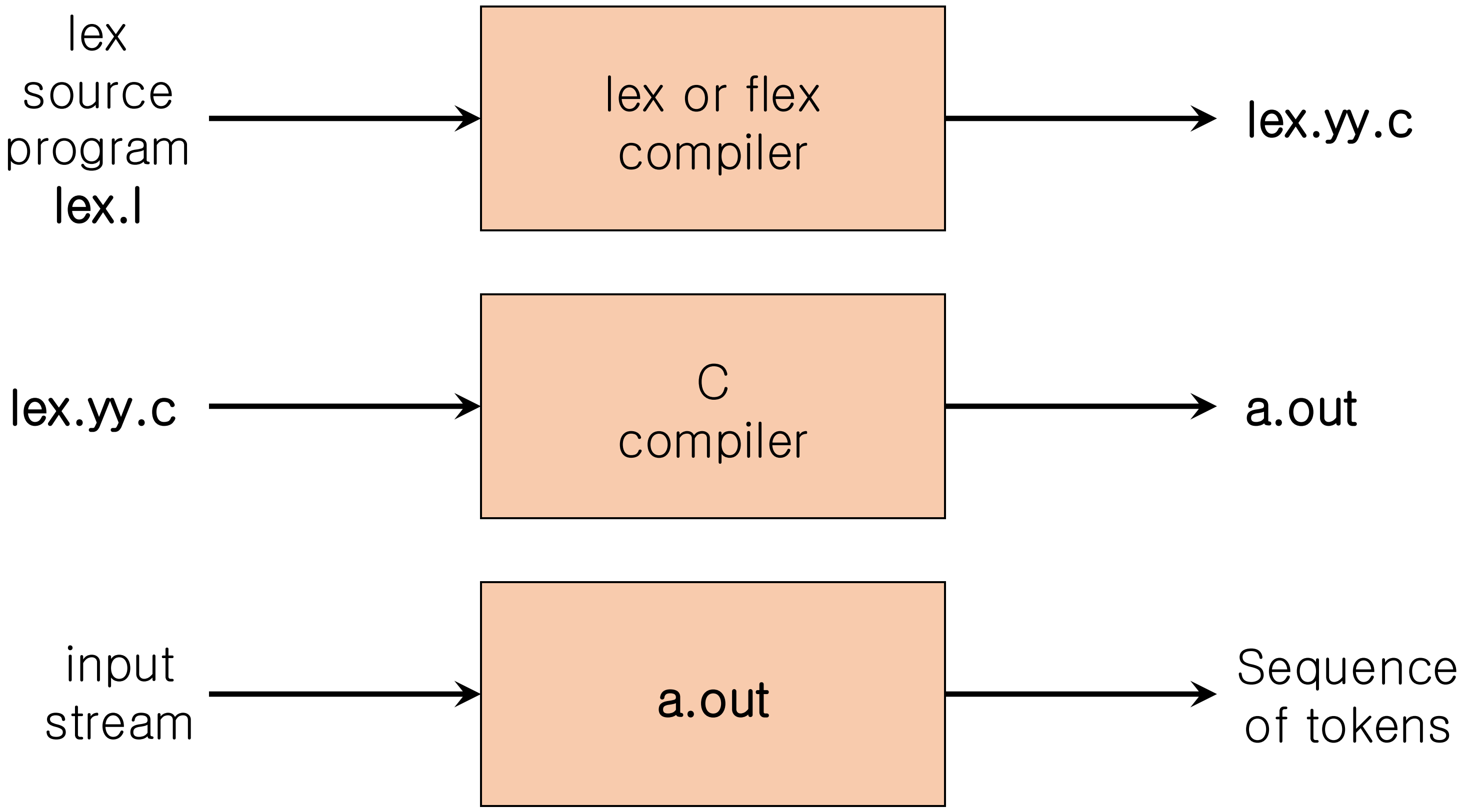
Lex : lexical analyzer generator

Lex : lexical analyzer generator

- The Lex and Flex scanner generators
  - *Lex* and its newer cousin *flex* are scanner generators
  - Systematically translate regular definitions into C source code for efficient scanning
  - Generated code is easy to integrate in C applications

Lex : lexical analyzer generator

- Create a lexical analyzer with Lex and Flex



Lex programs

- Form
  - Declarations : declare variables, manifest constants and regular definitions
  - Translation rules : Pattern {Action}
    - Pattern is regular expression
  - Auxiliary functions : whatever additional functions

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

YONSEI UNIVERSITY

Lex programs

- **Co-work with parser**
  - Parser calls lexical analyzer
  - It finds longest prefix of input that matches the pattern P_i and it executes action A_i
  - A_i returns to the parser.
    - If it does not (P_i is whitespace or comments), lexical analyzer proceed to next lexemes
  - Lexical Analyzer returns one value, **token name** to parser.

Lex programs

- Example 1

Translation rules

```
%{
#include <stdio.h>
%}
%%
[0-9]+  { printf("%s\n", yytext); }
.|\n    {  }
%%
main()
{
  yylex();
}
```

Contains the matching lexeme

Invokes the lexical analyzer

```
lex spec.l
gcc lex.yy.c -ll
./a.out < spec.l
```

Lex programs

- Example 2

Translation rules

Regular definition

```
%{
#include <stdio.h>
int ch = 0, wd = 0, nl = 0;
%}
delim     [ \t]+
%%
\n        { ch++; wd++; nl++; }
^{delim}  { ch+=yyleng; }
{delim}   { ch+=yyleng; wd++; }
.         { ch++; }
%%
main() {
  yylex();
  printf("%8d%8d%8d\n", nl, wd, ch);
}
```

Lex programs

- Example 3

Translation rules

Regular definitions

```
%{
#include <stdio.h>
%}
digit     [0-9]
letter    [A-Za-z]
id        {letter}({letter}|{digit})*
%%
{digit}+  { printf("number: %s\n", yytext); }
{id}      { printf("ident: %s\n", yytext); }
.         { printf("other: %s\n", yytext); }
%%
main() {
  yylex();
}
```

YONSEI UNIVERSITY

Lex programs

- **Conflict resolution and lookahead operator in Lex**
  - Conflict resolution
    - Input stream “if” or “<=“ ?
      - “<“ + “=“ and “<=“
        - Prefer longer prefix to the shorter one
      - Regular expressions for *if* and {id} match
        - Prefer the pattern listed first in lex program
  - The lookahead operator
    - IF(I,J) = 3 ?
      - 2-d array, named if, and assignment statement.
    - IF(condition) Then
      - Lex rule need to know if is a keyword if
      - IF / ₩(.* ₩) {letter}

```
x = 3 + 42 * (s - t)
```

A tokenizer splits the string into individual tokens

```
'x','=', '3', '+', '42', '*', '(', 's', '-', 't', ')'
```

Tokens are usually given names to indicate what they are. For example:

```
'ID','EQUALS','NUMBER','PLUS','NUMBER','TIMES',
'LPAREN','ID','MINUS','ID','RPAREN'
```

More specifically, the input is broken into pairs of token types and values. For example:

```
('ID','x'), ('EQUALS','='), ('NUMBER','3'),
('PLUS','+'), ('NUMBER','42), ('TIMES','*'),
('LPAREN','('), ('ID','s'), ('MINUS','-'),
('ID','t'), ('RPAREN',')'
```

```python
import ply.lex as lex

# List of token names.   This is always required
tokens = (
   'NUMBER',
   'PLUS',
   'MINUS',
   'TIMES',
   'DIVIDE',
   'LPAREN',
   'RPAREN',
)

# Regular expression rules for simple tokens
t_PLUS    = r'\+'
t_MINUS   = r'-'
t_TIMES   = r'\*'
t_DIVIDE  = r'/'
t_LPAREN  = r'\('
t_RPAREN  = r'\)'

# A regular expression rule with some action code
def t_NUMBER(t):
    r'\d+'
    t.value = int(t.value)
    return t

# Define a rule so we can track line numbers
def t_newline(t):
    r'\n+'
    t.lexer.lineno += len(t.value)

# A string containing ignored characters (spaces and tabs)
t_ignore  = ' \t'

# Error handling rule
def t_error(t):
    print("Illegal character '%s'" % t.value[0])
    t.lexer.skip(1)

# Build the lexer
lexer = lex.lex()
```

```
# Test it out
data = '''
3 + 4 * 10
  + -20 *2
'''

# Give the lexer some input
lexer.input(data)

# Tokenize
while True:
    tok = lexer.token()
    if not tok:
        break      # No more input
    print(tok)
```

When executed, the example will produce the following output:

```
$ python example.py
LexToken(NUMBER,3,2,1)
LexToken(PLUS,'+',2,3)
LexToken(NUMBER,4,2,5)
LexToken(TIMES,'*',2,7)
LexToken(NUMBER,10,2,10)
LexToken(PLUS,'+',3,14)
LexToken(MINUS,'-',3,16)
LexToken(NUMBER,20,3,18)
LexToken(TIMES,'*',3,20)
LexToken(NUMBER,2,3,21)
```

Questions?