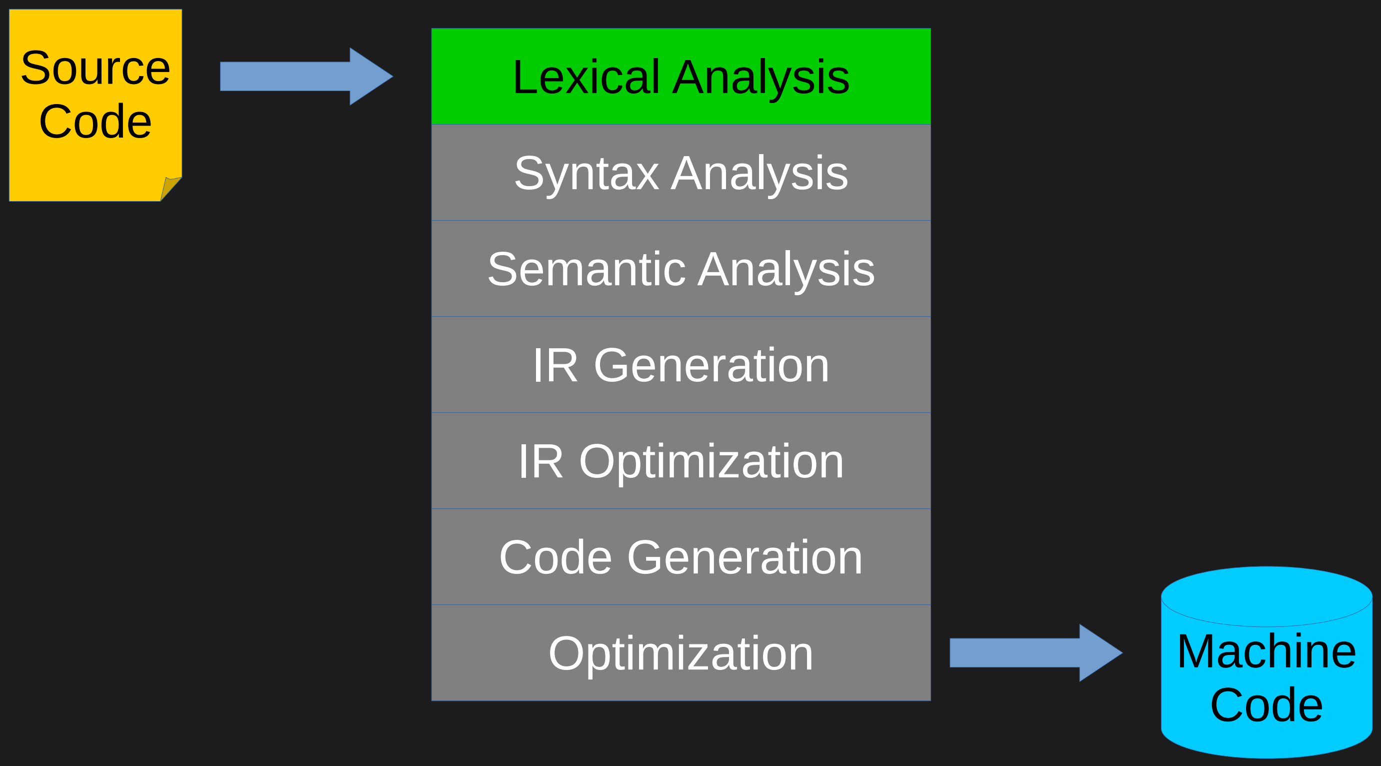
Lexical Analysis

Wei Wang

Where We Are


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

Textbook Chapters

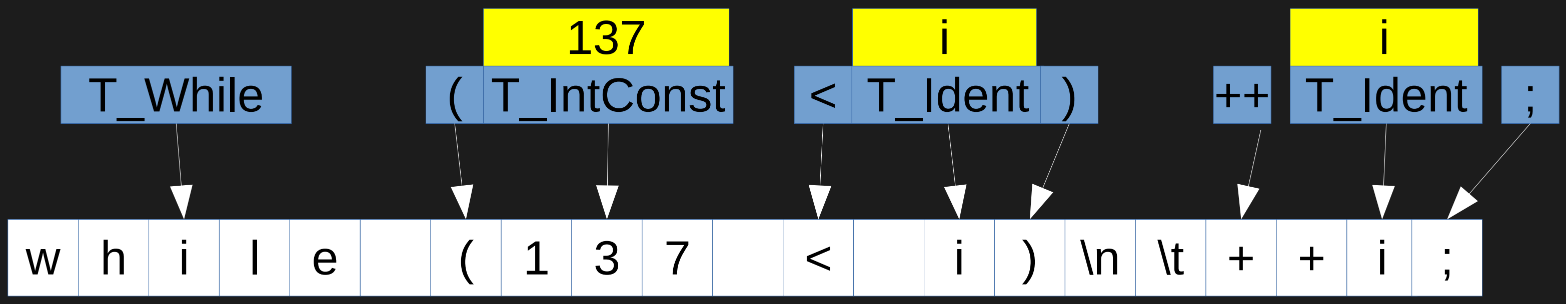
- Dragon Book
  - Chapters 3.1, 3.3, 3.4, and 3.8

An Example of Scanning

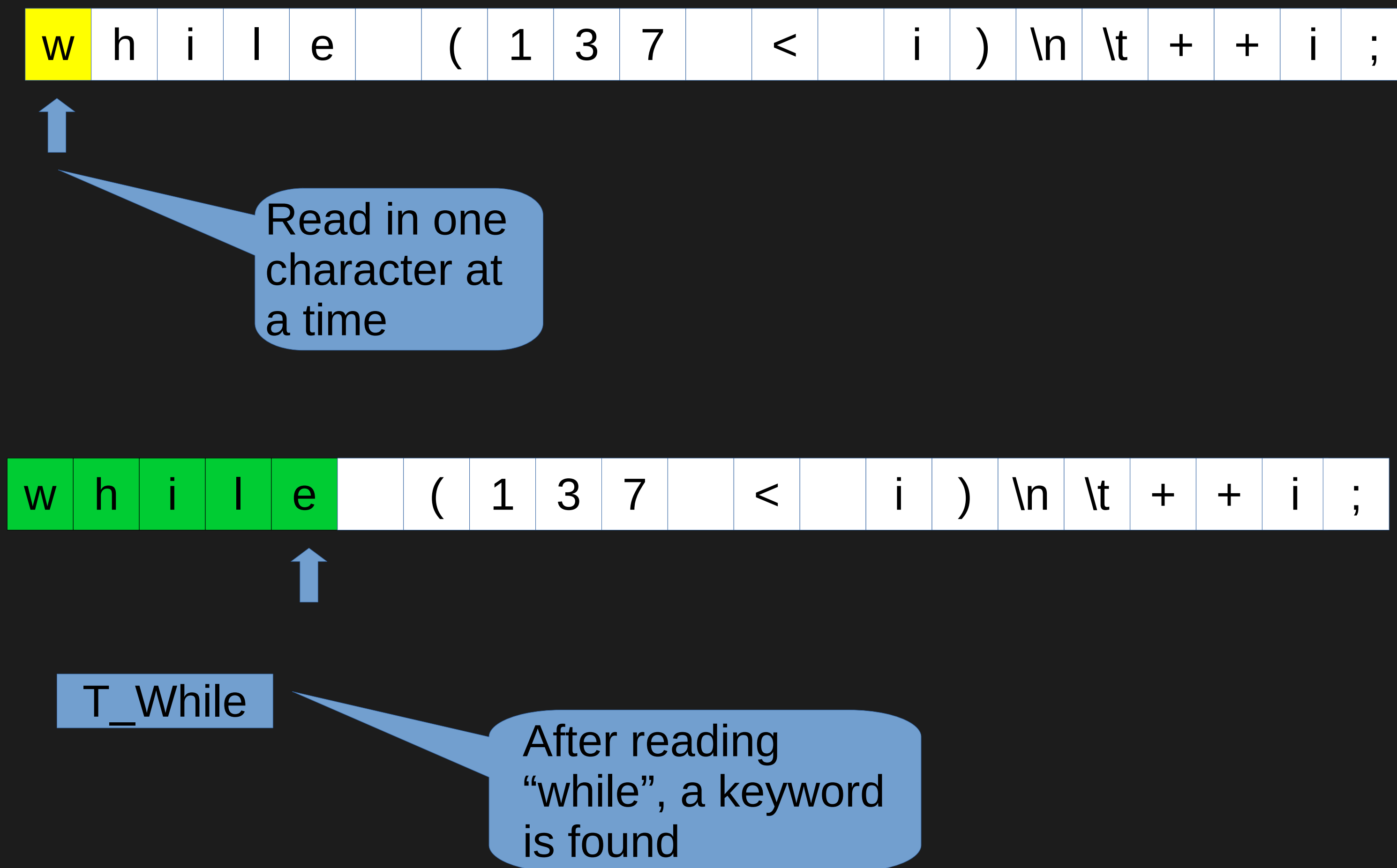
Lexical Analysis

- Lexical analysis recognize the basic lexemes of source code.
- For example,

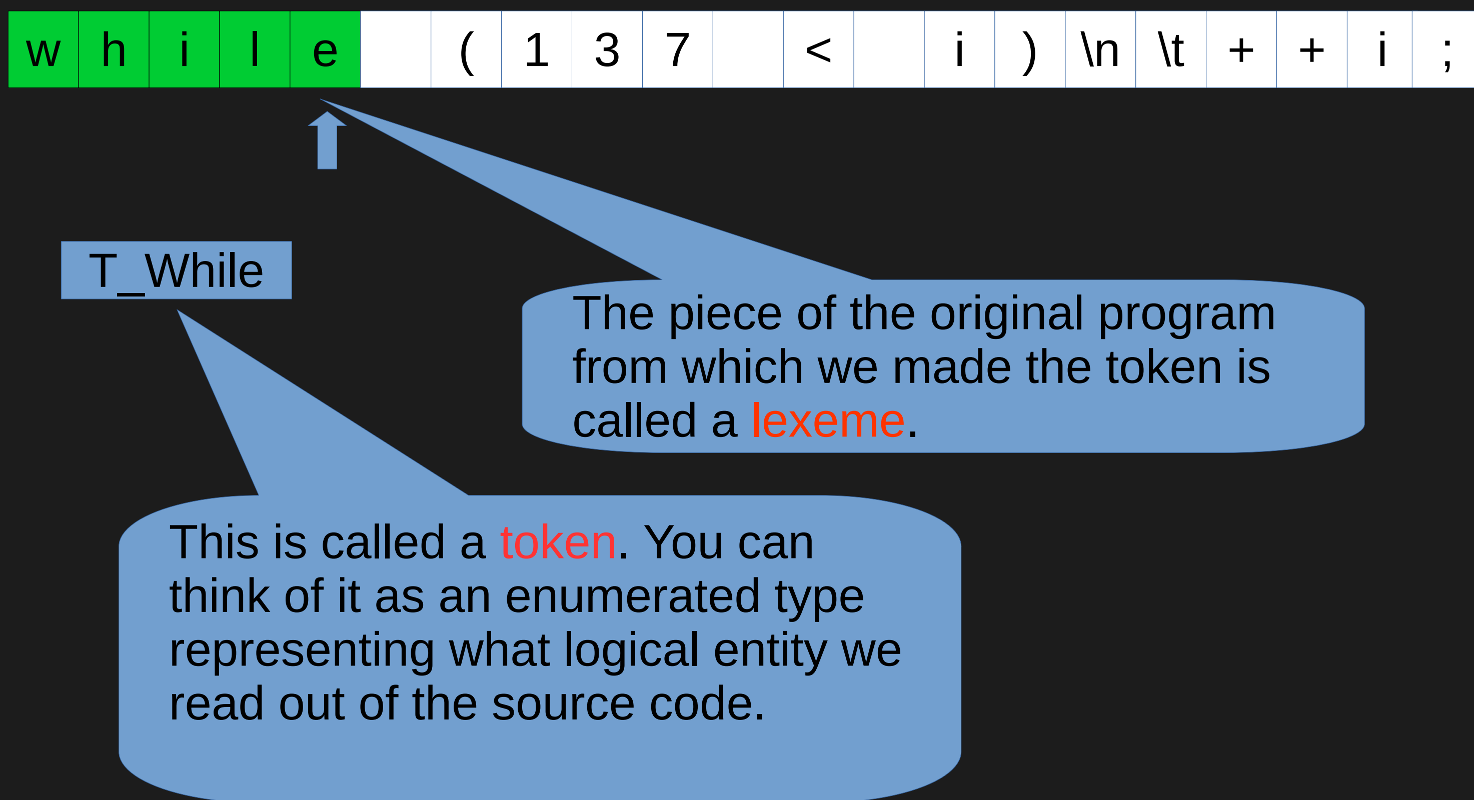
```
while (137 < i)
      ++i;
```



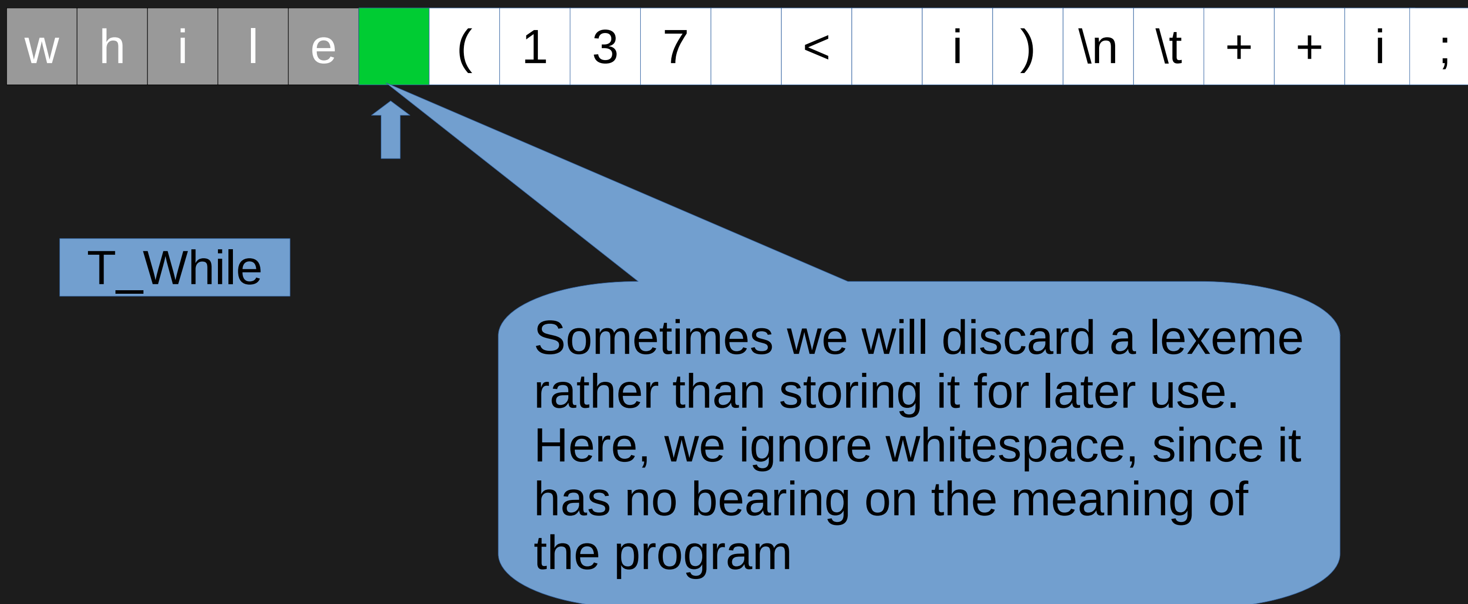
Scanning a Source File



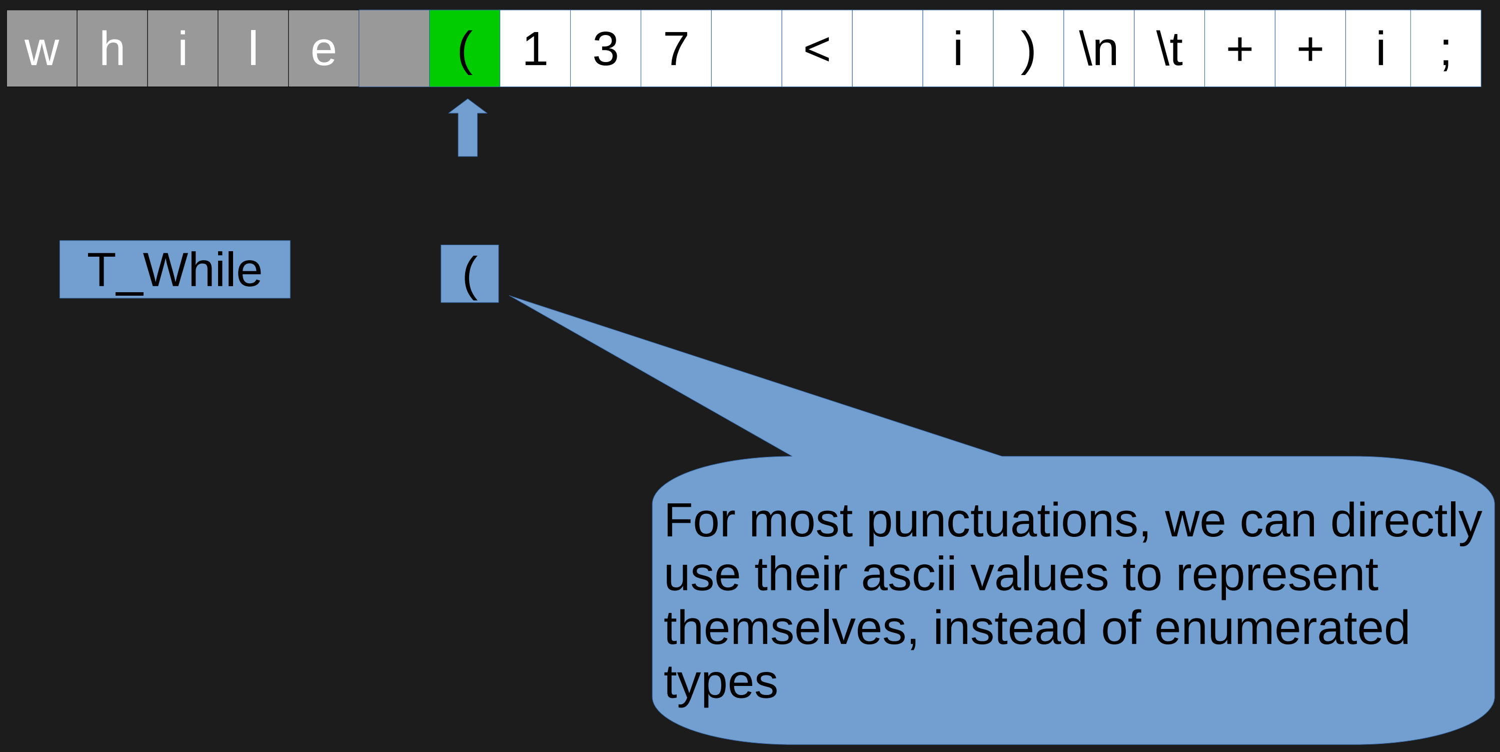
Scanning a Source File cont.



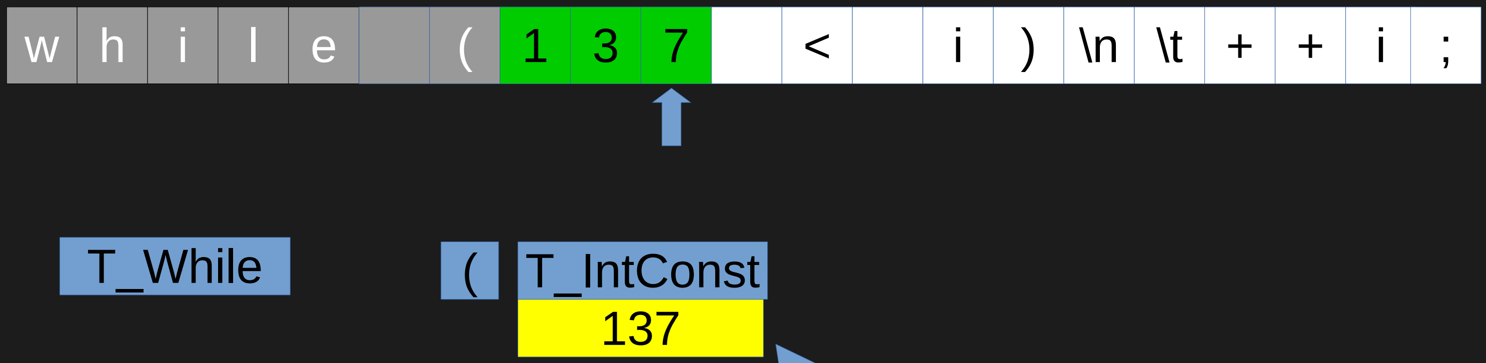
Scanning a Source File cont.



Scanning a Source File cont.

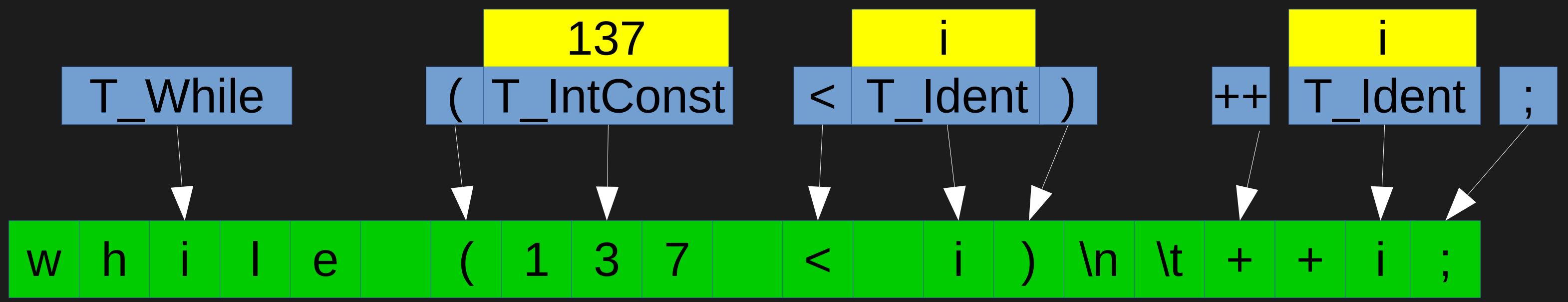


Scanning a Source File cont.



Some tokens can have attributes that store extra information about the token. Here we store which integer is represented.

Scanning a Source File cont.



The tokens after all lexemes are scanned. Note that identifiers also have associated values, where are the variable names.

Goals of Lexical Analysis

- Convert from physical description of a program into sequence of tokens.
  - Each token represents one logical piece of the source file – a keyword, the name of a variable, etc.
  - Each token is associated with a lexeme.
  - The actual text of the token: “137,” “int,” etc.
- Each token may have optional attributes.
  - Extra information derived from the text – perhaps a numeric value.
- The token sequence will be used in the parser to recover the program structure

Scanning is Hard

Choosing Tokens

- What Tokens are Useful Here?

```
for (int k = 0; k < myArray[5]; ++k) {
         cout << k << endl;
}
```

```
for { int } << ; = < ( [ ) ] ++

Identifier

IntegerConstant
```

Choosing Good Tokens cont.

- Very much dependent on the language.
- Typically:
  - Give keywords their own tokens.
  - Give different punctuation symbols their own tokens.
  - Group lexemes representing identifiers, numeric constants, strings, etc. into their own groups.
- Discard irrelevant information (whitespace, comments)

Scanning is Hard

- FORTRAN: Whitespace is irrelevant

```
DO 5 I = 1,25
DO5I = 1.25
```

- Can be difficult to tell when to partition input.

Scanning is Hard cont.

- C++: Nested template declarations

```
vector <vector<int>> myVector
```

  - Or,

```
vector < vector < int >> myVector
```

  - Or,

```
(vector < (vector < (int >> myVector)))
```

  - Again, can be difficult to determine where to split.

Scanning is Hard cont.

- PL/1: Keywords can be used as identifiers.

```
IF THEN THEN THEN = ELSE; ELSE ELSE = IF
```

- Can be difficult to determine how to label lexemes.

Associating Lexemes with Tokens

Lexemes and Tokens

- Tokens give a way to categorize lexemes by what information they provide.
- Some tokens might be associated with only a single lexeme:
  - Tokens for keywords like if and while probably only match those lexemes exactly.
- Some tokens might be associated with lots of different lexemes:
  - All variable names, all possible numbers, all possible strings, etc.

Sets of Lexemes

- Idea: Associate a set of lexemes with each token.
- We might associate the “number” token with the set { 0, 1, 2, …, 10, 11, 12, … }
- We might associate the “string” token with the set { "", "a", "b", "c", … }
- We might associate the token for the keyword while with the set { while }.

Expressing the Sets of Lexemes

- For most languages, we use Regular Expressions (RE) to express the sets of lexemes.
- Regular expressions are a family of descriptions that can be used to capture certain languages (the regular languages).
- Often provide a compact and human-readable description of the language.
- Used as the basis for numerous software systems, including the flex tool we will use in this course.
- Recall the REs are recognized with DFA and NFA.

Examples of REs in Compilers

- An integer has only 0, 1, 2, …, 9
  - That is, the RE for integer is [0-9]+
- An identifier (ID) must starts with a letter, and may contain letters, numbers and underscore
  - RE for ID is then [A-Za-z]([A-Za-z0-9_)*

A Challenge in Scanning

- How do we determine which lexemes are associated with each token?

Associating Lexemes with Tokens

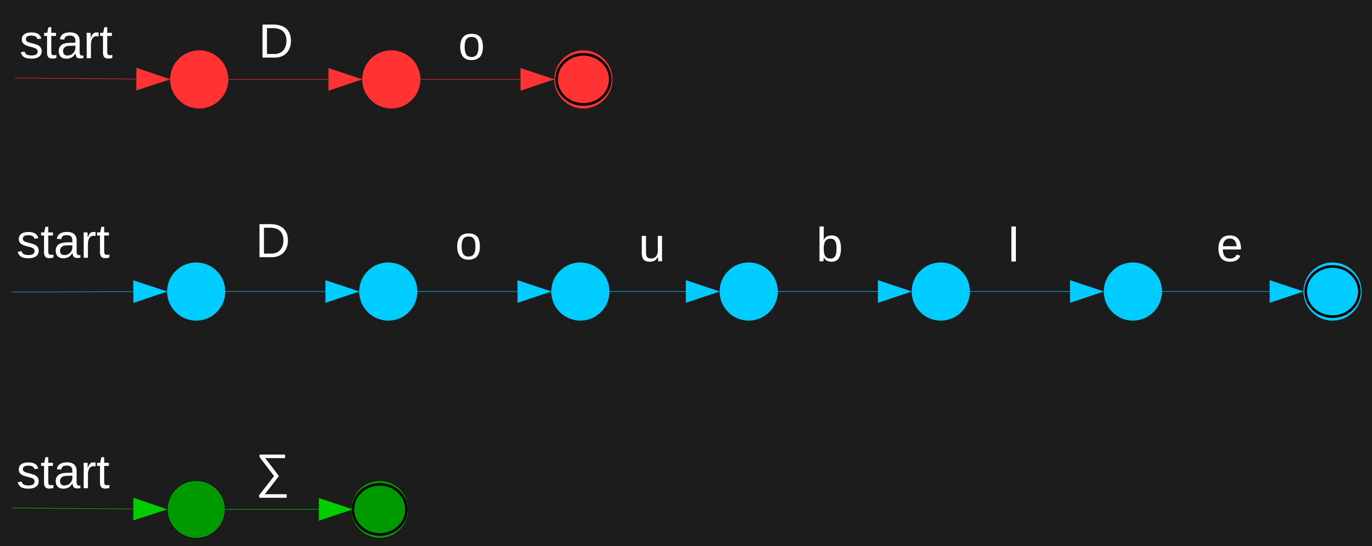
- Is “for” an identifier or a keyword?
- Is “fort” an identifier or a keyword?
- How can a scanner tell the difference?
- Conflict Resolution:
  - Assume all tokens are specified as regular expressions.
  - Algorithm: Left-to-right scan.
  - Tiebreaking rule one: Longest match.
    - Always match the longest possible prefix of the remaining text

Implementing Longest Match

- 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.

Implementing Longest Match cont.

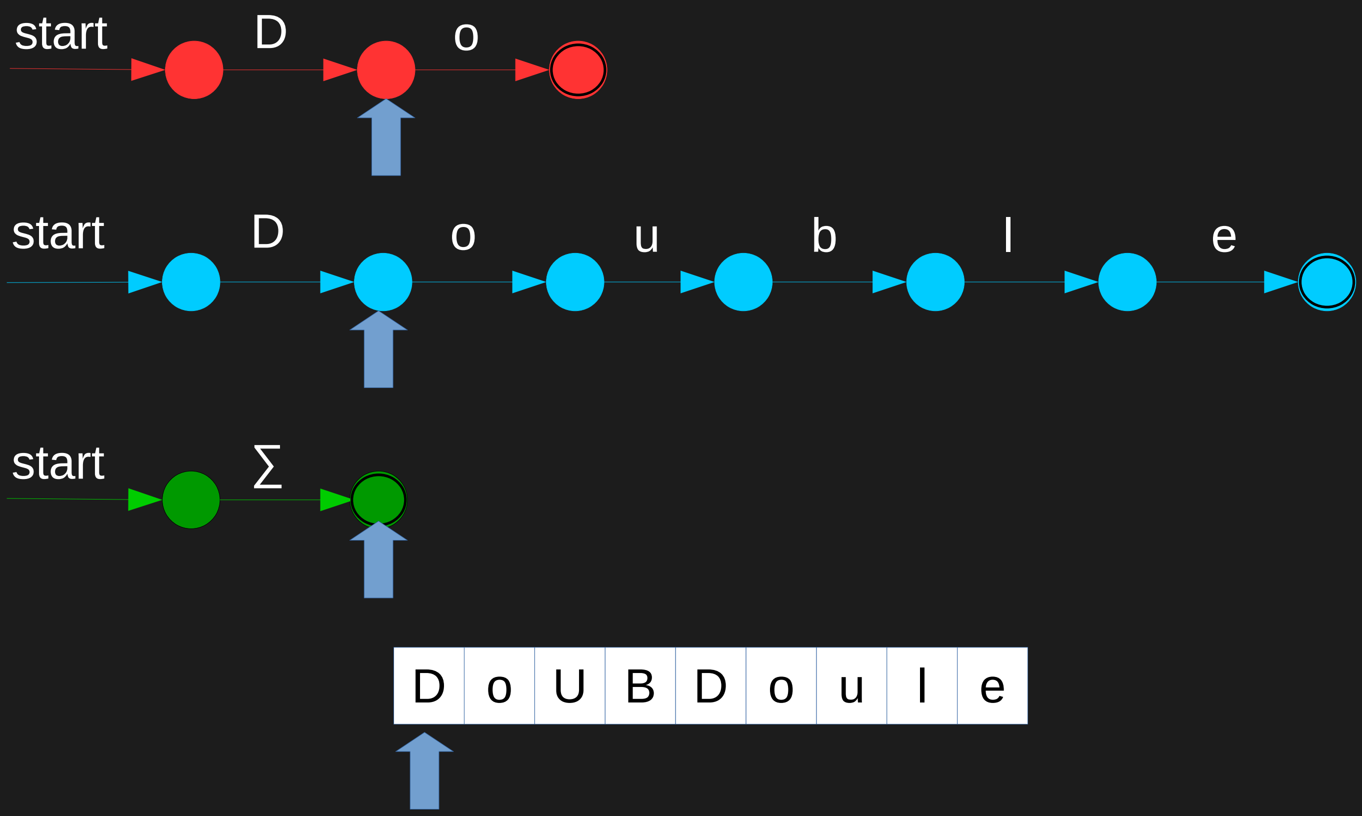
```
T_DO      do
T_Double  double
T_Mystery [A-Za-z]
```



D	o	U	B	D	o	u	l	e

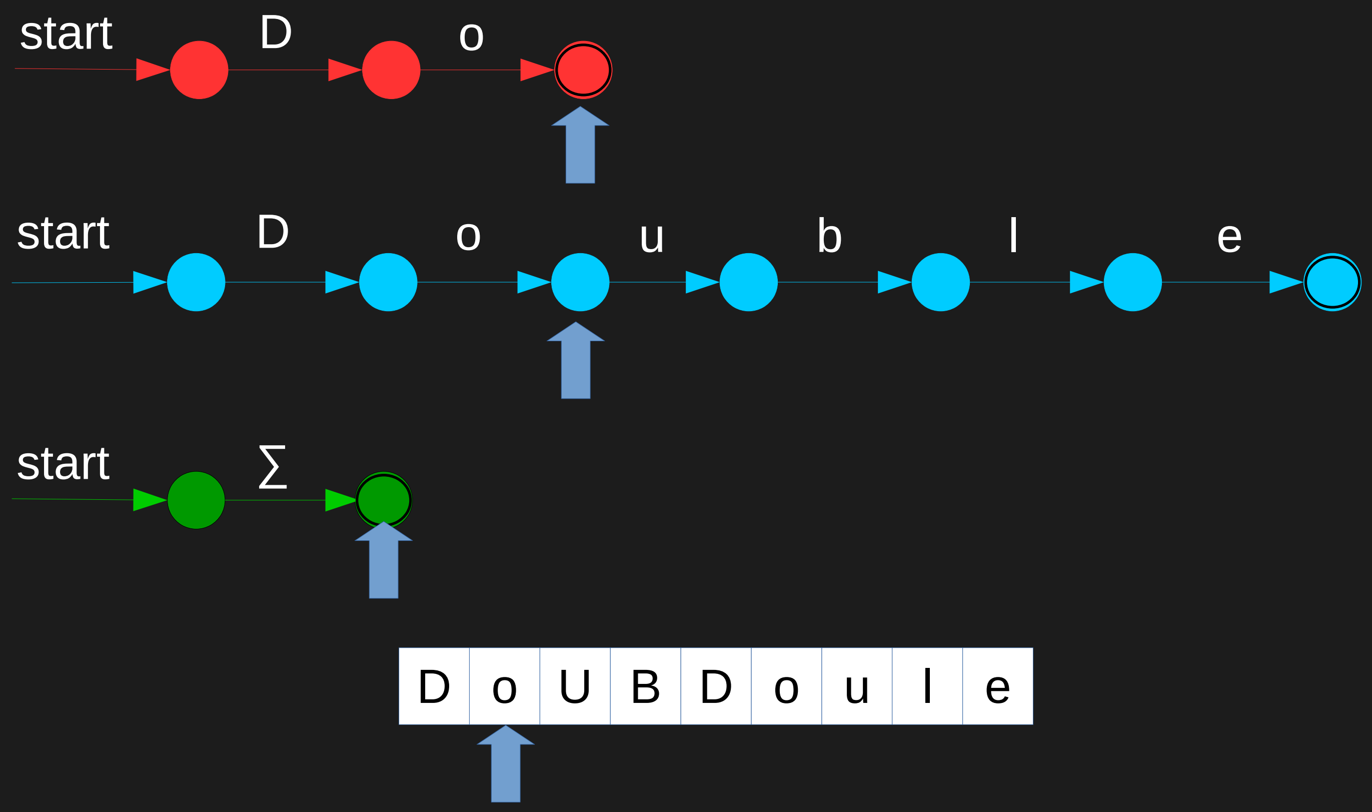
Implementing Longest Match cont.

```
T_DO       do
T_Double   double
T_Mystery  [A-Za-z]
```



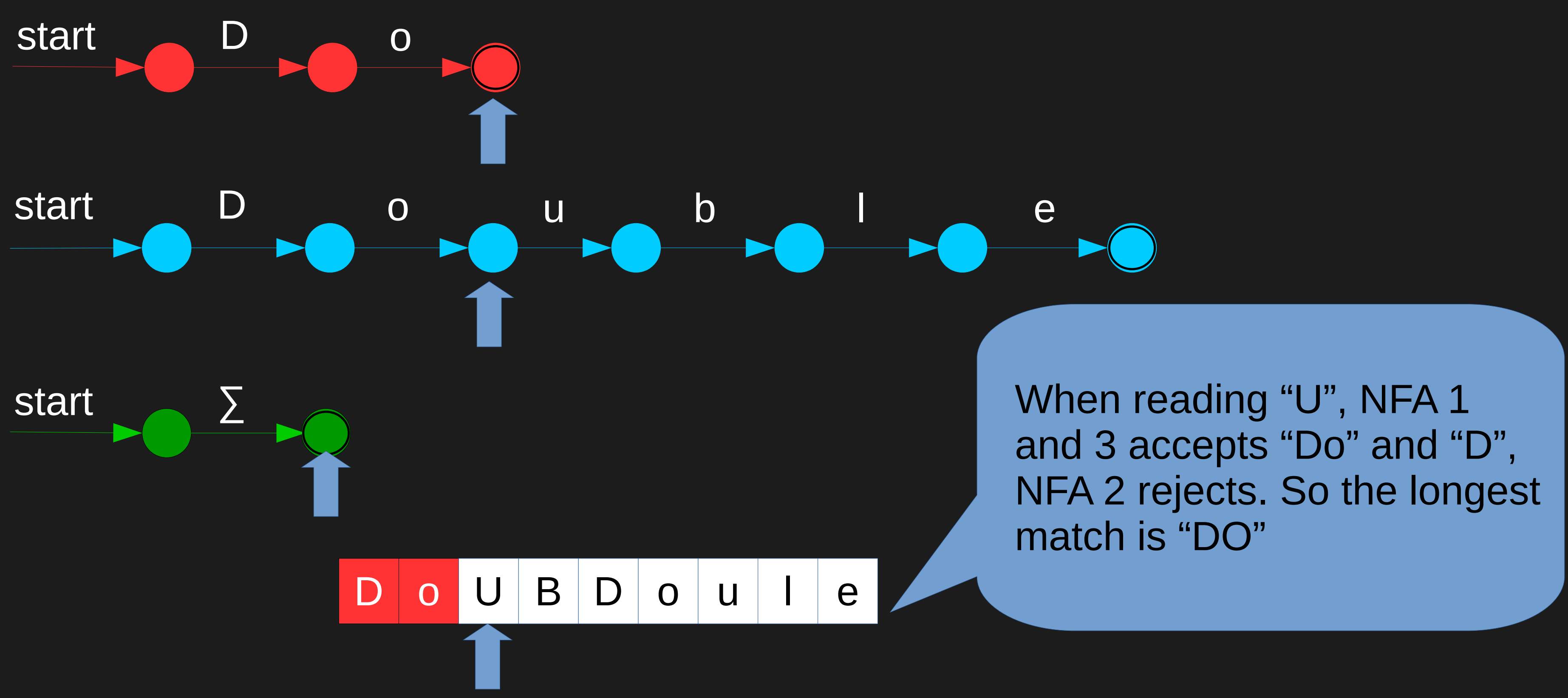
Implementing Longest Match cont.

```
T_DO       do
T_Double   double
T_Mystery  [A-Za-z]
```



Implementing Longest Match cont.

```
T_DO       do
T_Double   double
T_Mystery  [A-Za-z]
```



More Tiebreaking

- When two regular expressions apply, choose the one with the greater “priority.”
- Simple priority system: pick the rule that was defined first.
- E.g., “int ” matches both T_Identifier and T_Int. If we give T_Int higher priority, then “int” is considered to be T_Int.
- In Flex, rules defined earlier has higher priority.

No Rule Matches

- We know what to do if multiple rules match.
- What if nothing matches?
- Trick: Add a “catch-all” rule with lowest priority that matches any character and reports an error.

Summary of Conflict Resolution

- Construct an automaton for each regular expression.
- Scan the input, keeping track of the last known match in each automaton.
  - It is possible to merge these automata into on deterministic automaton.
- Break ties by choosing higher-precedence matches.
- Have a catch-all rule to handle errors.

Acknowledgement

- This lectures is based on the Compiler slides of Keith Schwarz.