Compiler Construction

Instructor: เกียรติกูล เจียรนัยธนะกิจ

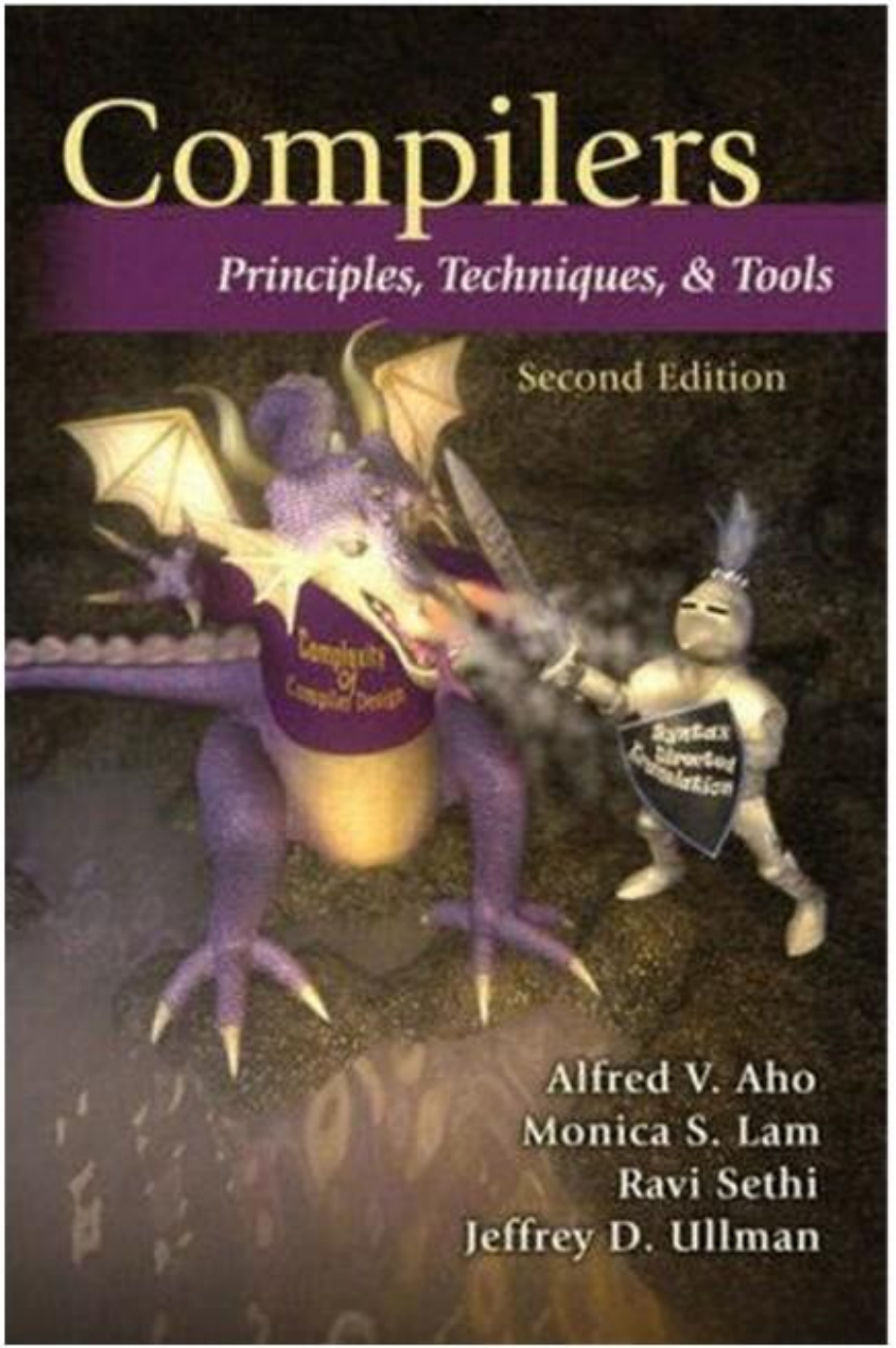
Kietikul Jearanaitanakij

E-mail: kietikul@yahoo.com // primary
kjkietik@kmitl.ac.th // secondary

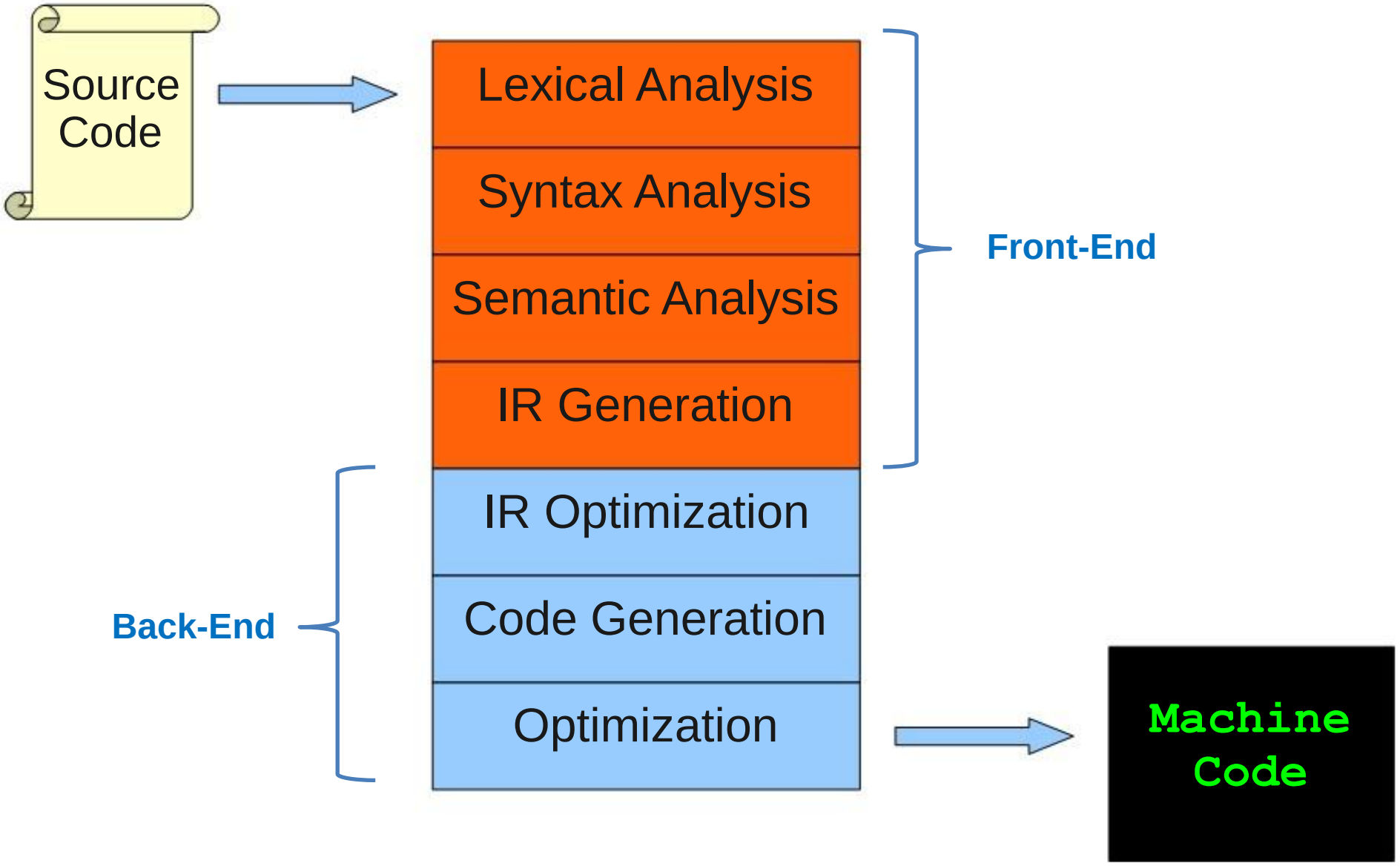
Grading criteria:

Assignments	50 %	
- Lexical analyzer		20 %
- Parser		20 %
- Report		10 %
Final exam	50%	

Slides: Adapted from Compiler course @ Stanford University


Compilers
Principles, Techniques, & Tools
Second Edition
Alfred V. Aho
Monica S. Lam
Ravi Sethi
Jeffrey D. Ullman

The Structure of a Modern Compiler

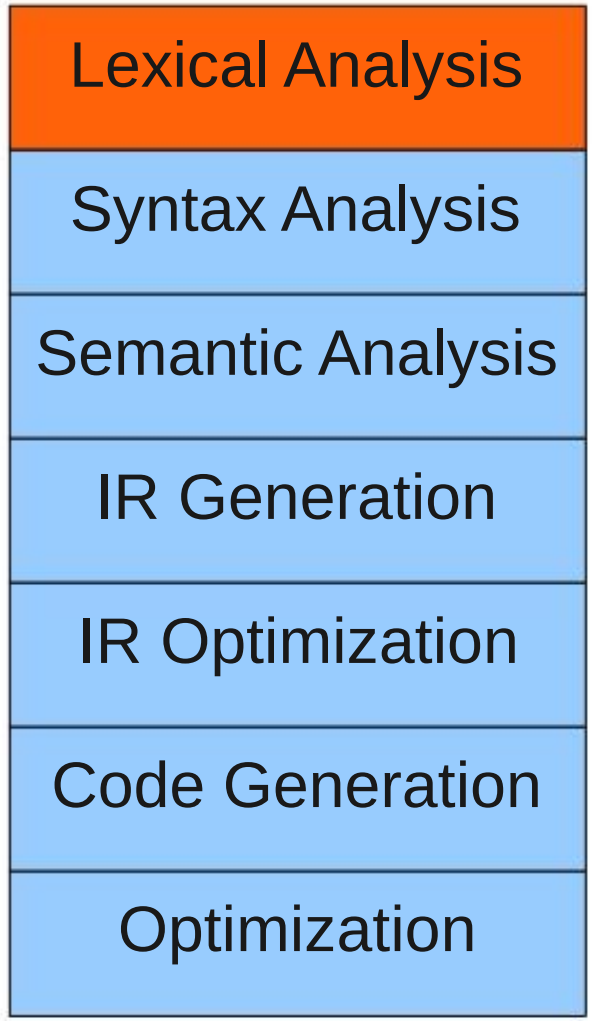


```
while (y < z) {
    int x = a + b;
    y += x;
}
```

"Identify tokens in the source code."

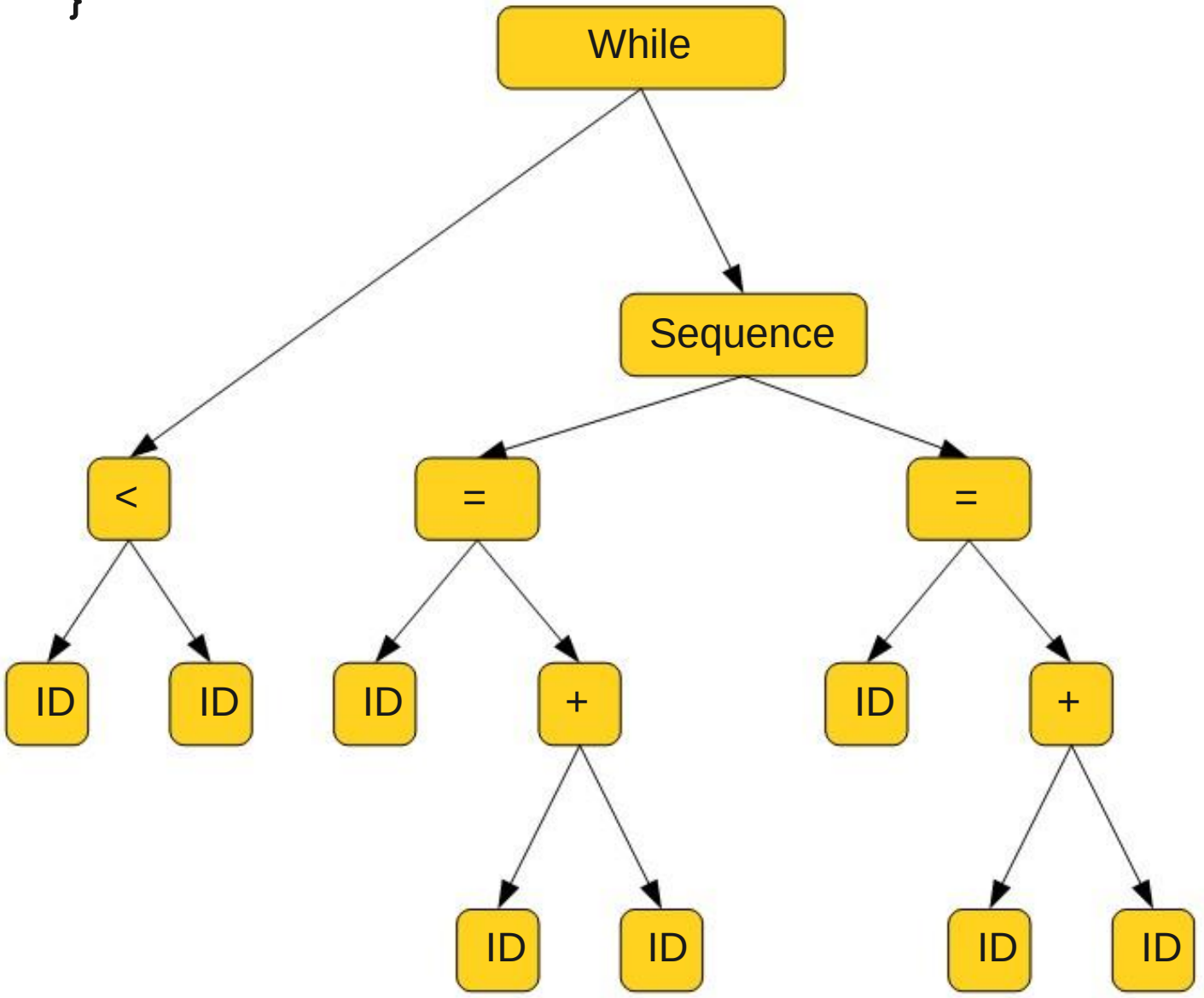
```
T_While
T_LeftParen
T_Identifier y
T_Less
T_Identifier z
T_RightParen
T_OpenBrace
T_Int
T_Identifier x
T_Assign
T_Identifier a
T_Plus
T_Identifier b
T_Semicolon
T_Identifier y
T_PlusAssign
T_Identifier x
T_Semicolon
T_CloseBrace
```

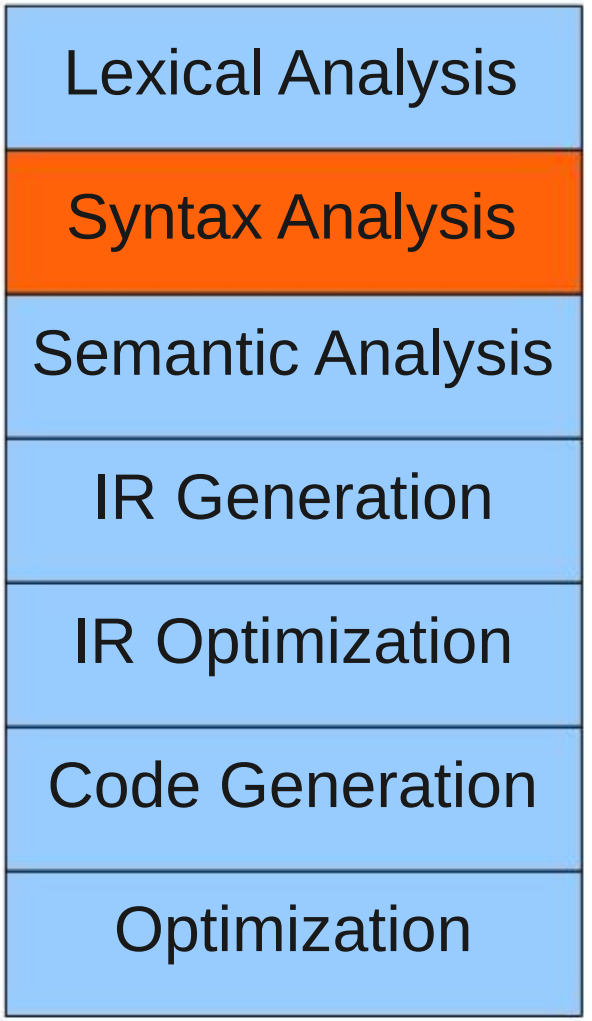
A list of "Tokens"



```
while (y < z) {
    int x = a + b;
    y += x;
}
```

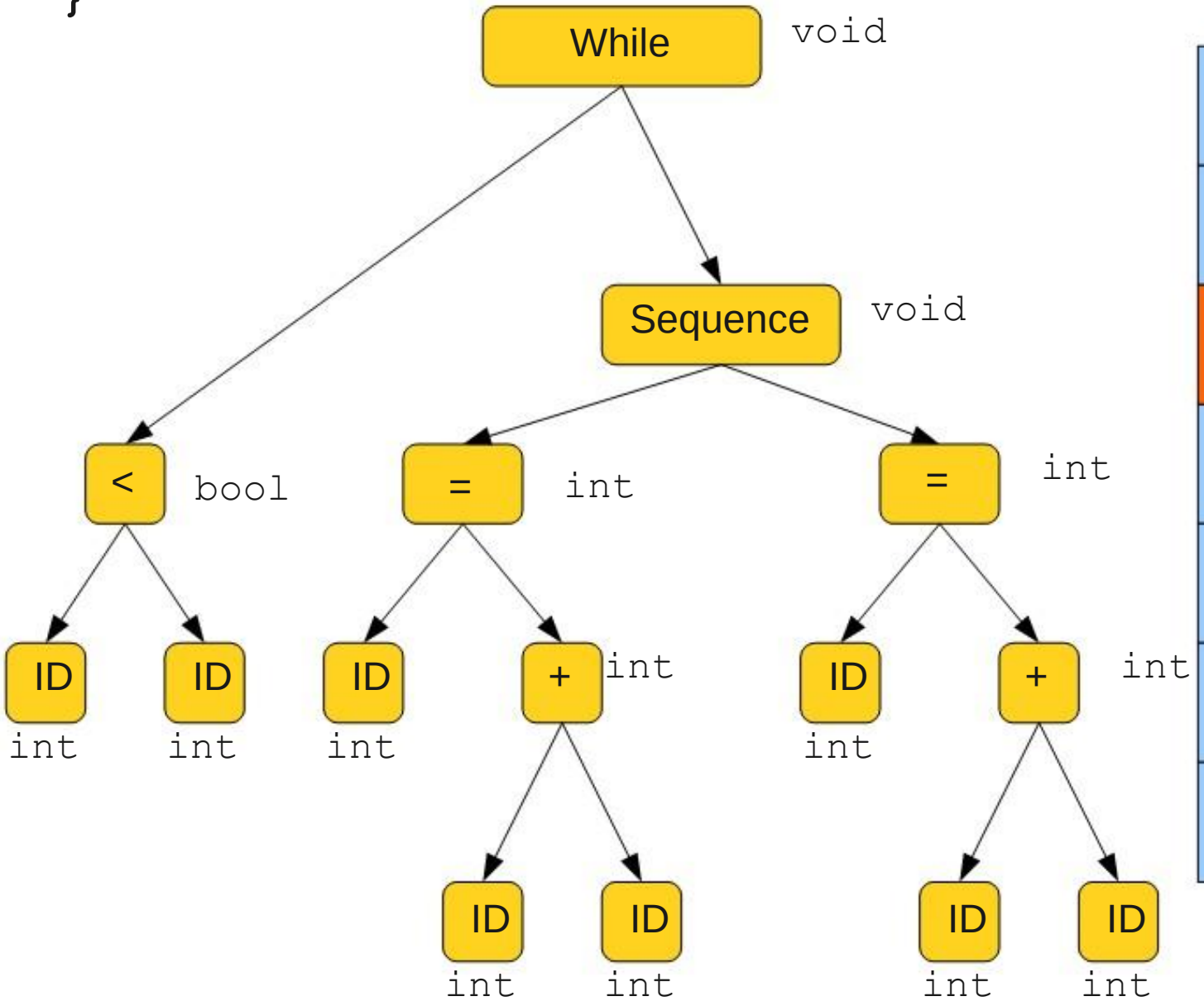
"Identify how those tokens relate to each other."





```
while (y < z) {
    int x = a + b;
    y += x;
}
```

"Identify the meaning of those relations."

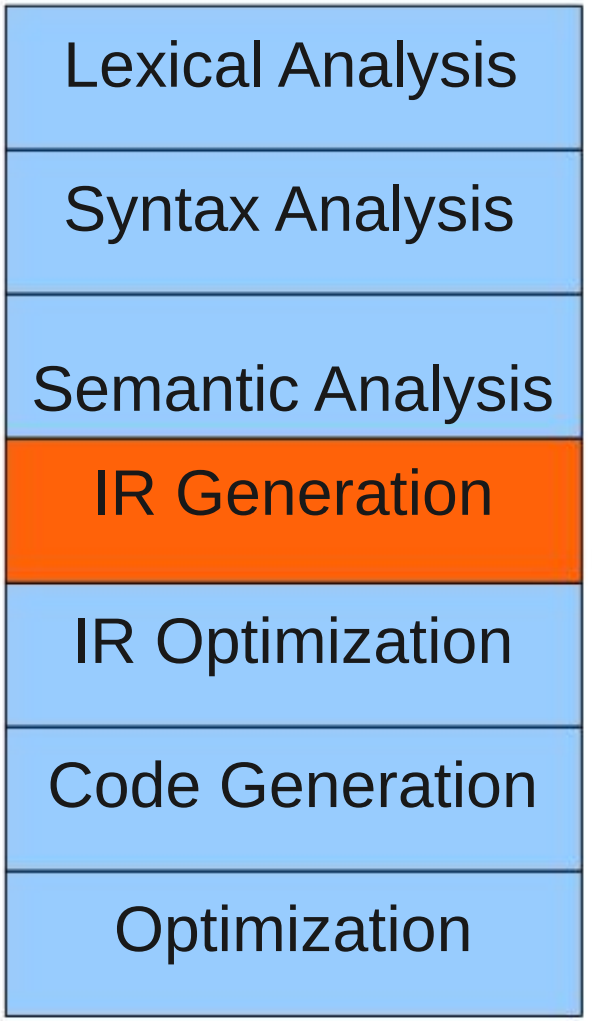


- Lexical Analysis
- Syntax Analysis
- **Semantic Analysis**
- IR Generation
- IR Optimization
- Code Generation
- Optimization

```
while (y < z) {
    int x = a + b;
    y += x;
}
```

"Generate the assembly-like code."

```
Loop: _t1 = y < z
      if not _t1 goto Exit
      x   = a + b
      y   = x + y
      goto Loop
Exit:
```

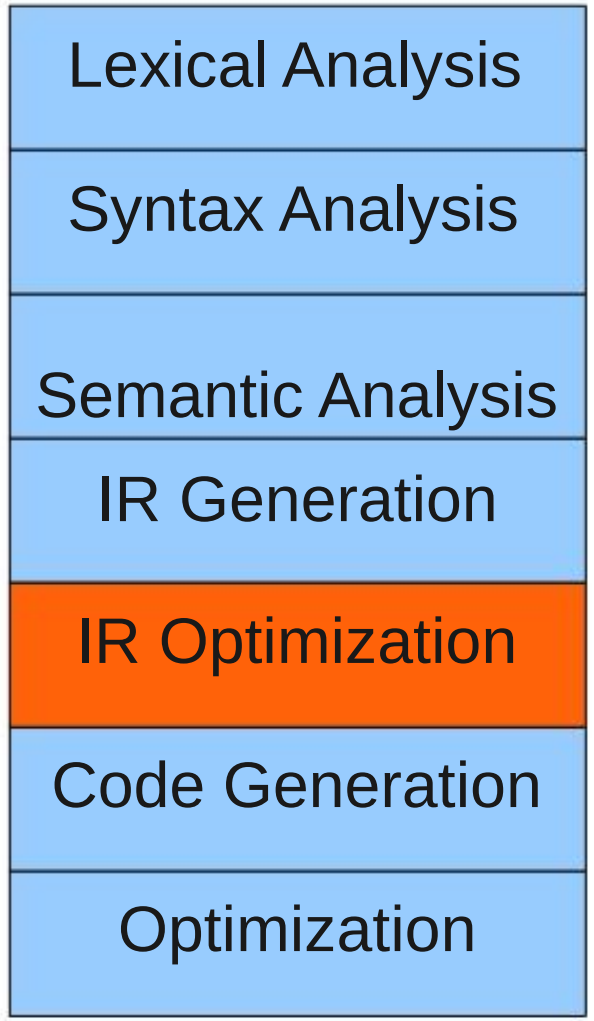


IR : Intermediate Representation

```
while (y < z) {
    int x = a + b;
    y += x;
}
```

"Optimize the assembly-like code."

```
      x   = a + b
Loop: _t1 = y < z
      if not _t1 goto Exit
      y   = x + y
      goto Loop
Exit:
```

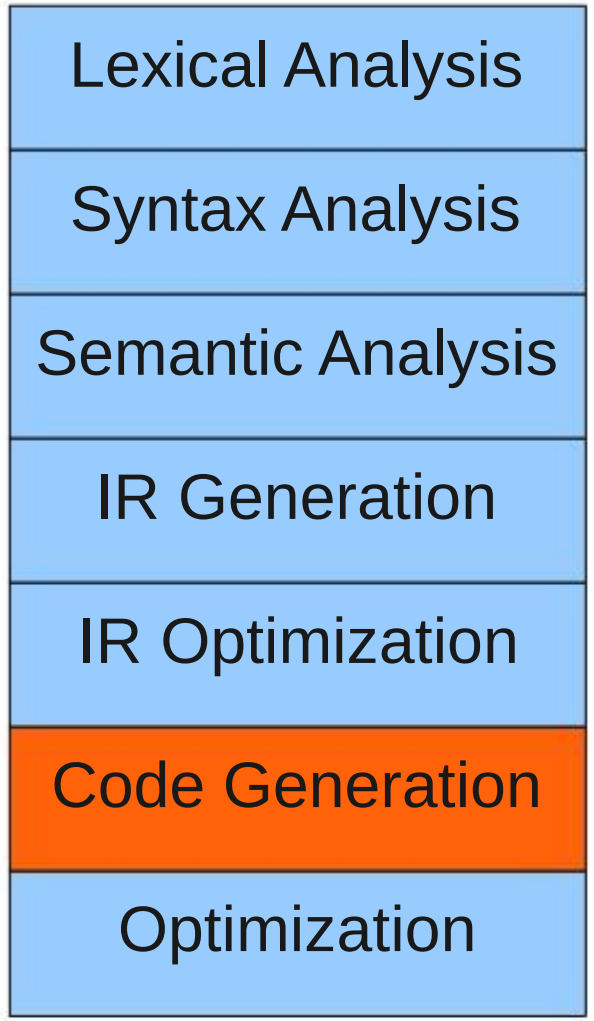


```
while (y < z) {
    int x = a + b;
    y += x;
}
```

"Generate the machine code."

```
        add $1, $2, $3
Loop:   slt $6, $4, $5
        beq $6, Exit
        add $4, $4, $1
        b   Loop

 Exit:
```

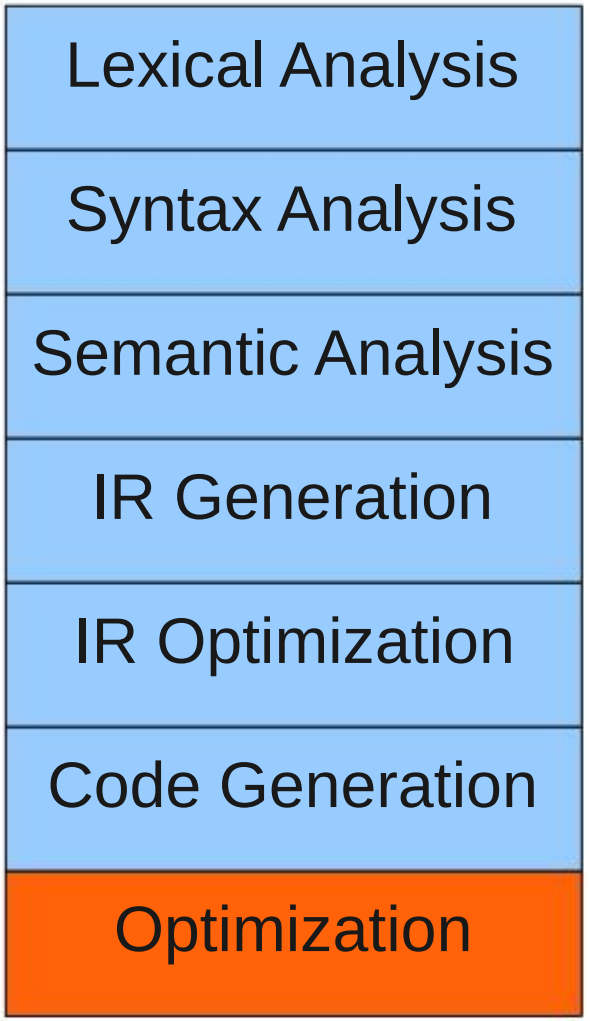


```
while (y < z) {
    int x = a + b;
    y += x;
}
```

"Optimize the machine code."

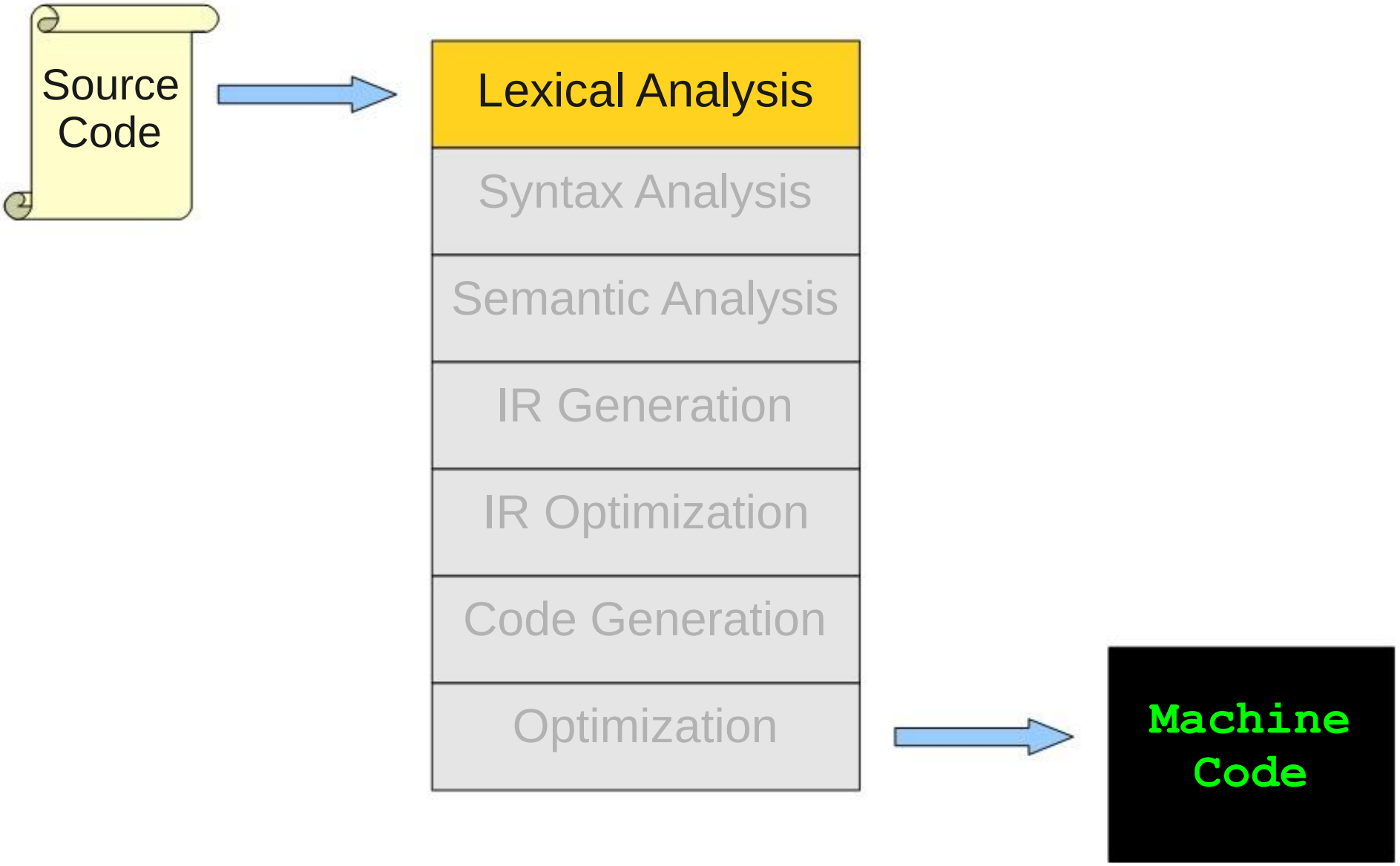
```
        add $1, $2, $3
Loop:   blt $4, $5, Exit
        add $4, $4, $1
        b   Loop

Exit:
```



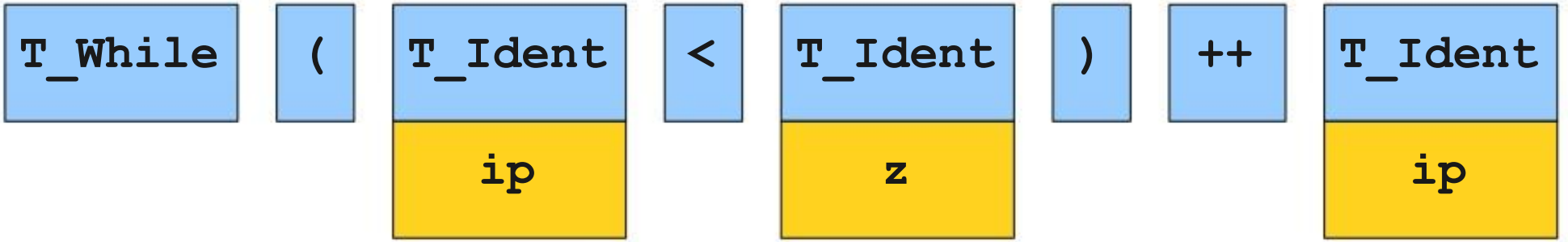
Lexical Analysis

Where We Are


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

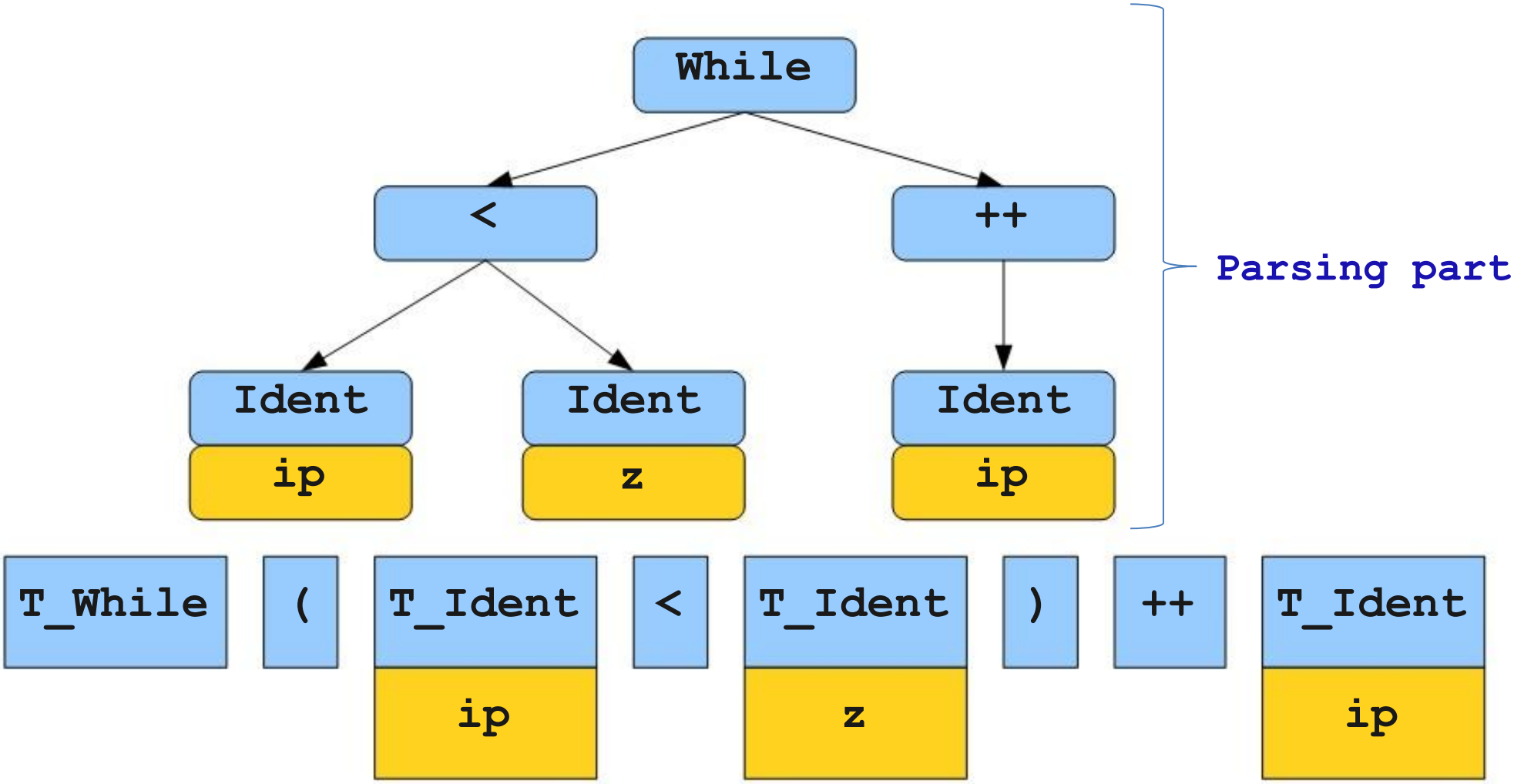
Why do Lexical Analysis?

- Dramatically simplify parsing.
  - Eliminate whitespace, comments.
  - Provide input for the parsing part.



w	h	i	l	e		(	i	p		<		z	)	\n	\t	+	+	i	p	;

```
while (ip < z)
  ++ip;
```

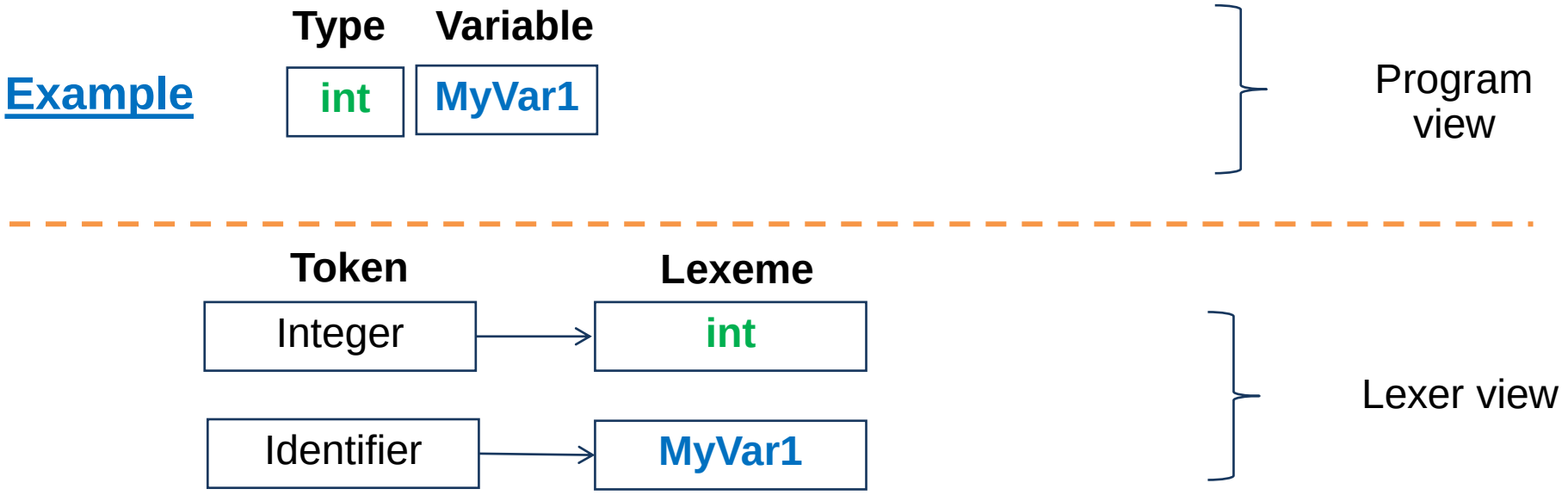


w	h	i	l	e		(	i	p		<		z	)	\n	\t	+	+	i	p	;

```
while (ip < z)
++ip;
```

Goals of Lexical Analysis

- Convert from physical description of a program into sequence of **tokens**.



- The token stream will be used in the parser to create the syntax tree.

Challenges in Lexical Analysis

- How to partition the program into lexemes?

```
DO 5 I = 1,25

Equivalent to

DO

  // repeat loop for I = 1 to 25

5:
```

```
DO 5 I = 1.25

Equivalent to

     DO5I = 1.25

Whitespace is irrelevant
```

- FORTRAN requires look ahead symbol to partition the program into lexemes.

FORTRAN (Formula Translation) Originally developed by IBM

Not-so-Great Moments in Scanning

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

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

- Can be difficult to determine how to label lexemes.

PL/I (Programming Language One) Originally developed by IBM

Credit to Prof. Alex Aiken

Choosing Good Tokens

- Very much dependent on the language.
- Typical kinds of token:
  - Reserve words (If, Then, Else, For, While, Do, etc.).
  - Punctuation symbols (; “ ‘ < > + - * / % ! [ ] { } && || , etc.)
  - Identifier
  - Number (Integer, floating point, etc.)
  - Discard irrelevant information (whitespace, comments)
- Token should be recognized in maximal munch manner

  Example:

  dot should be recognized as an Identifier “dot”

  (Not a reserved word DO and an identifier “t”)

Implementing Maximal Munch Lexer

- Represent all tokens as regular expressions.

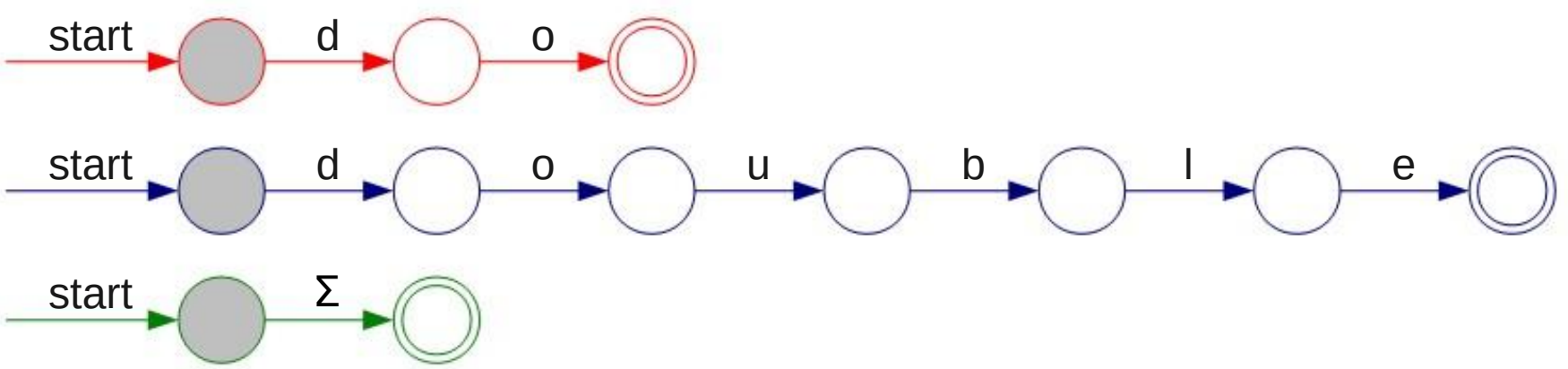
Example

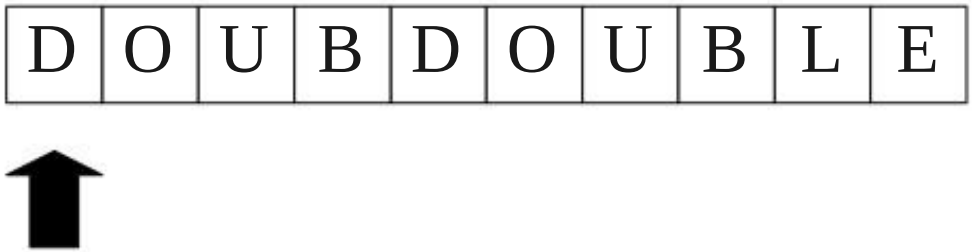
Token	Regular expression
T_Do	do
T_IDEN	[a-z][a-zA-Z0-9]*

- Convert all regular expressions to NFAs.
- Scan input from Left to Right
- Run all NFAs in parallel, keeping track of the last match.
- When all automata get stuck, report the longest match and restart the search at that point.

Implementing Maximal Munch

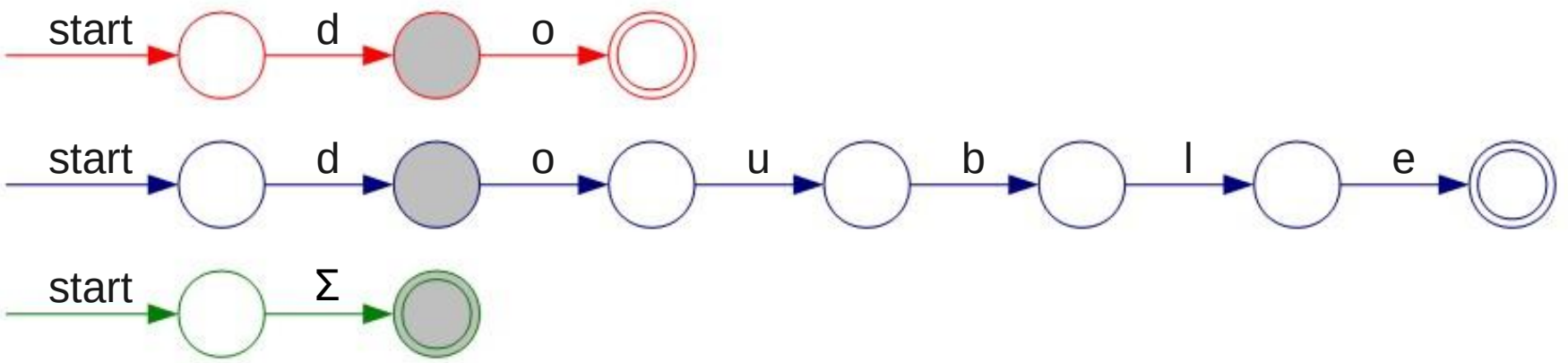
```
T_Do          do
T_Double      double
T_Mystery     [A-Za-z]
```

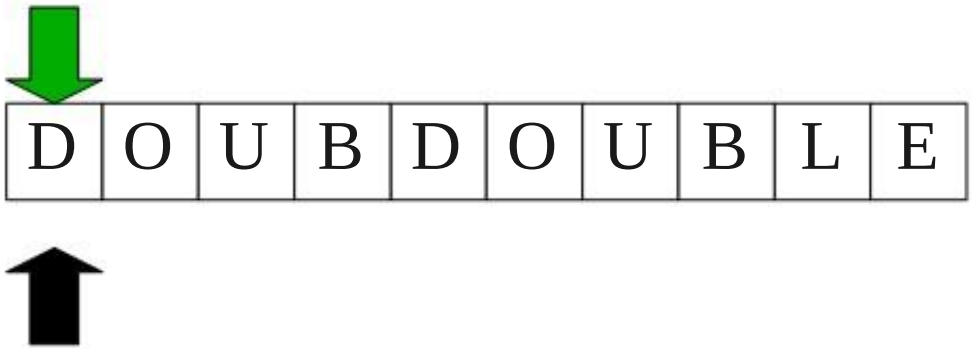




Implementing Maximal Munch

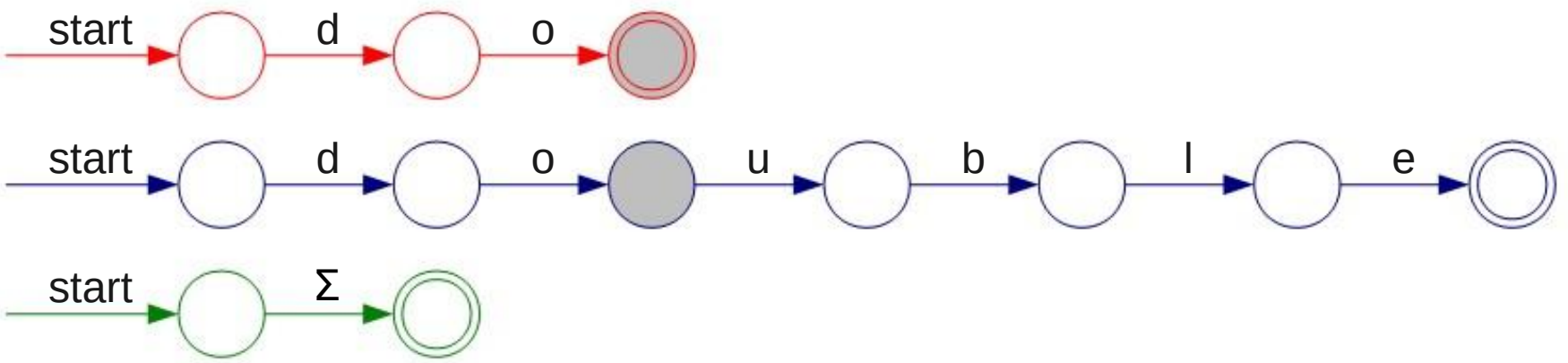
```
T_Do          do
T_Double      double
T_Mystery     [A-Za-z]
```

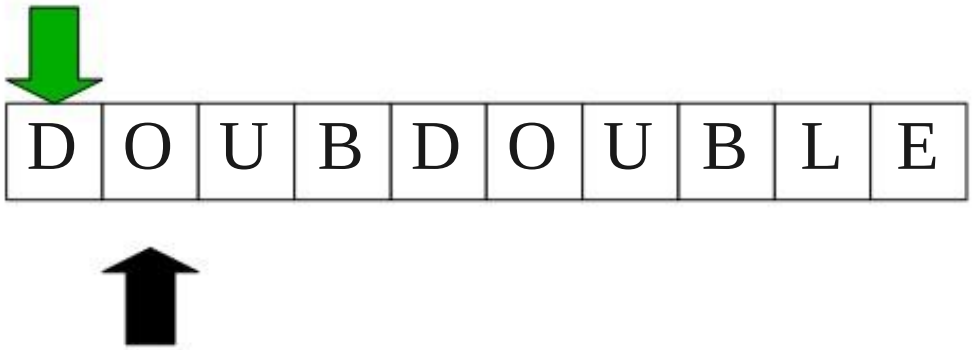




Implementing Maximal Munch

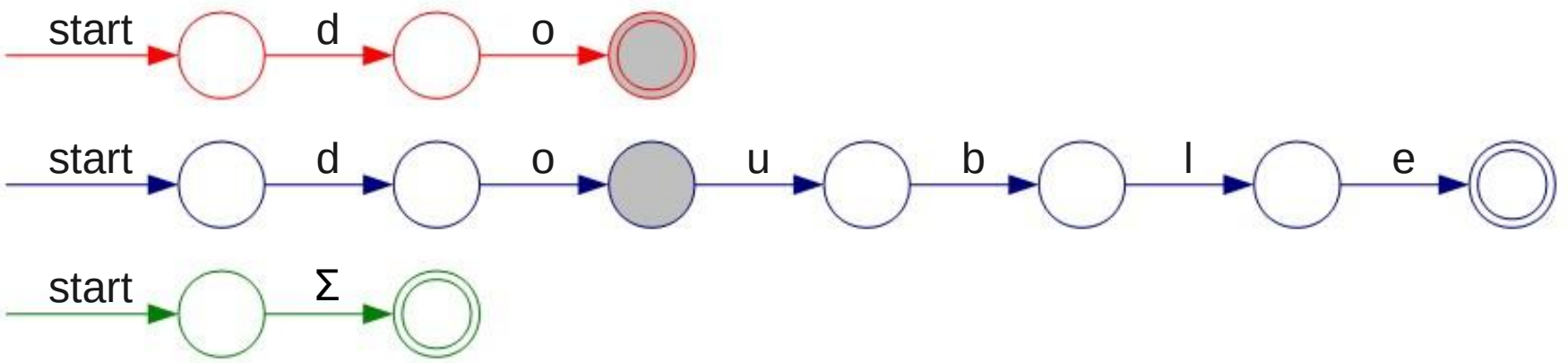
```
T_Do          do
T_Double      double
T_Mystery     [A-Za-z]
```

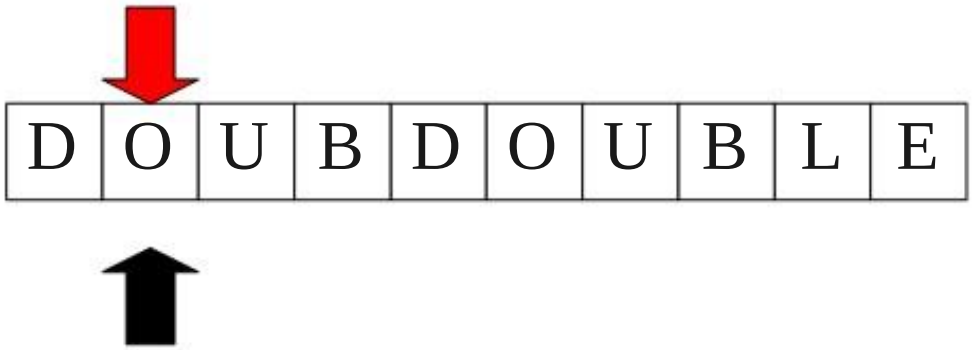




Implementing Maximal Munch

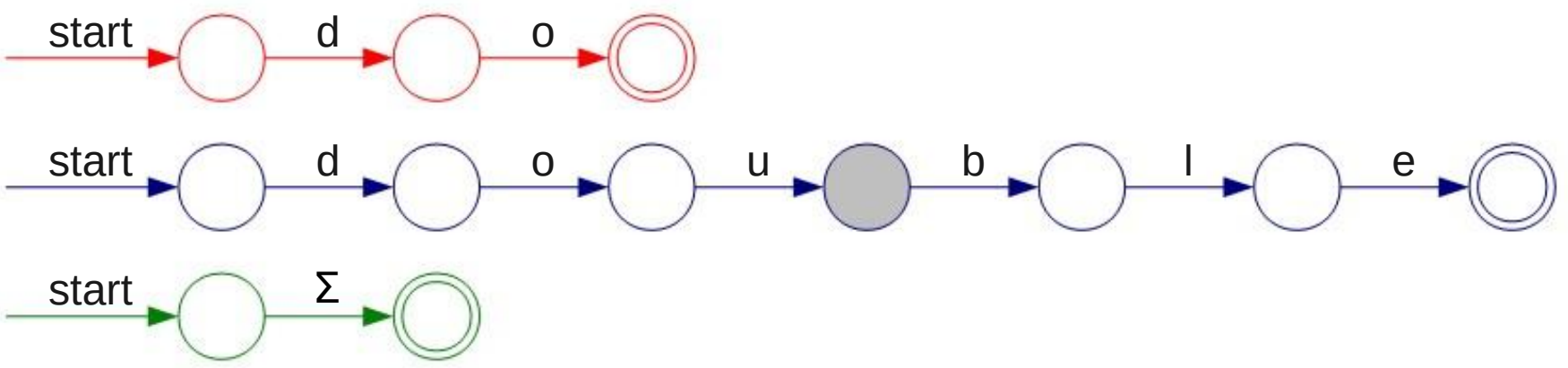
```
T_Do          do
T_Double      double
T_Mystery     [A-Za-z]
```

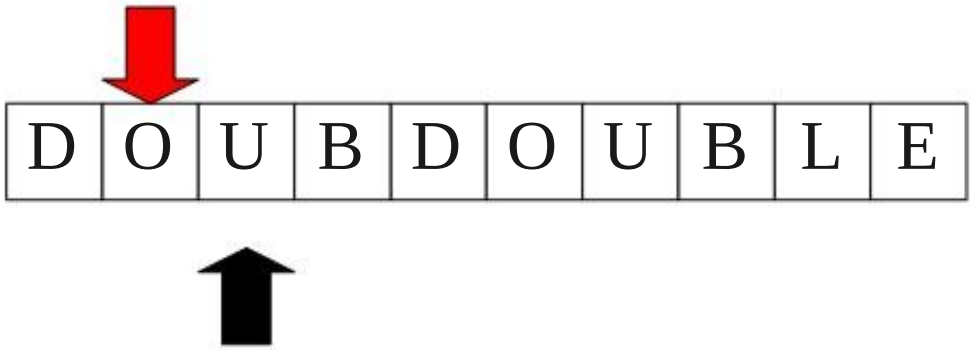




Implementing Maximal Munch

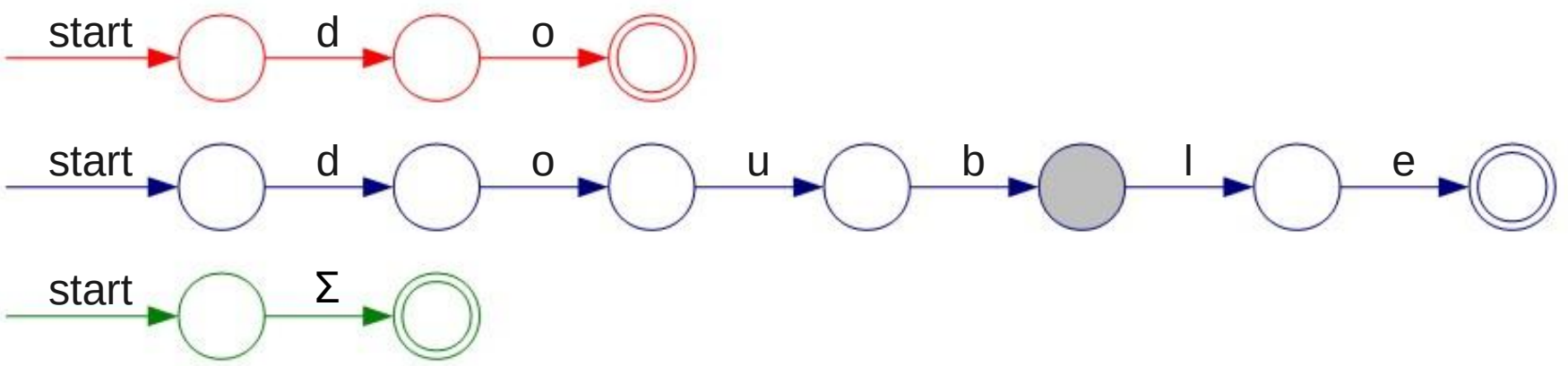
```
T_Do          do
T_Double      double
T_Mystery     [A-Za-z]
```

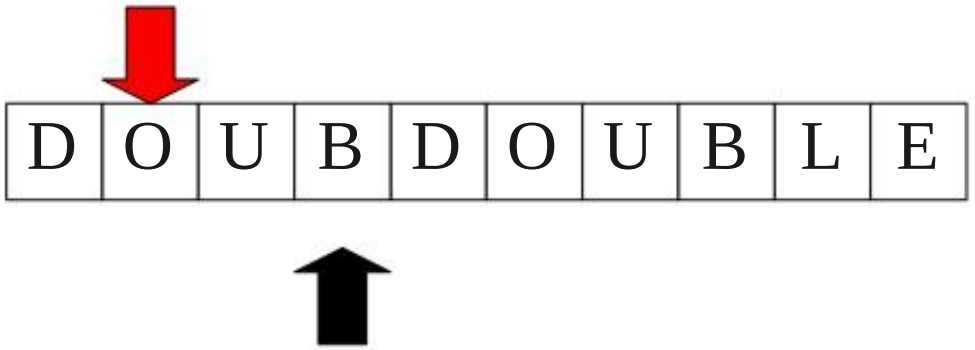




Implementing Maximal Munch

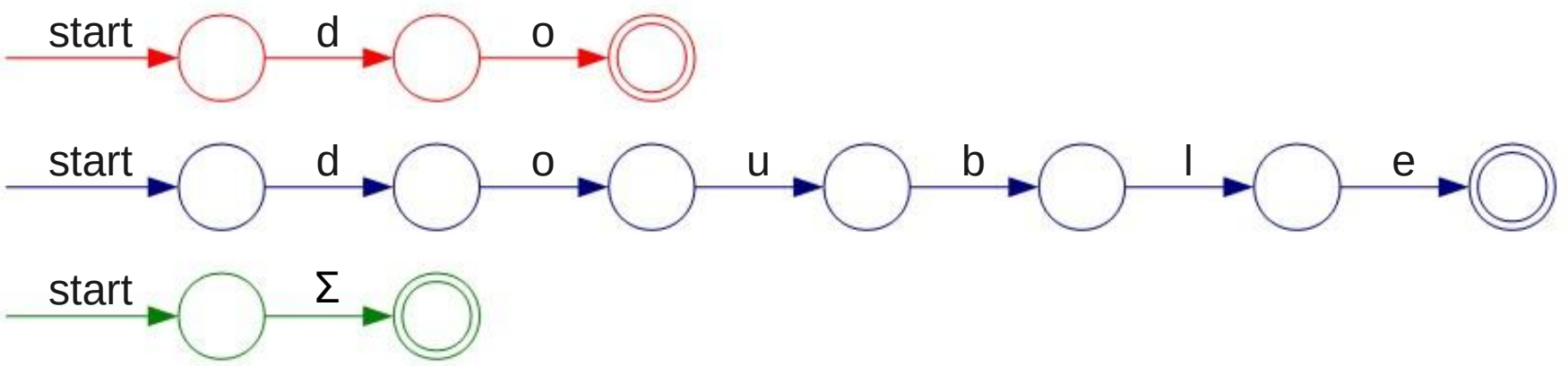
```
T_Do          do
T_Double      double
T_Mystery     [A-Za-z]
```

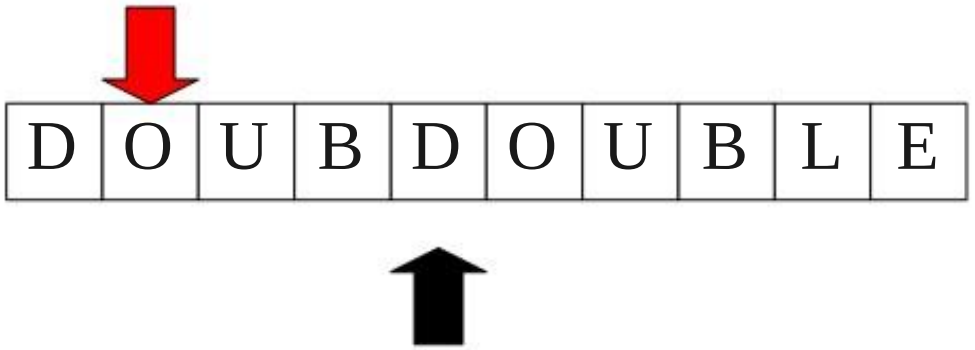




Implementing Maximal Munch

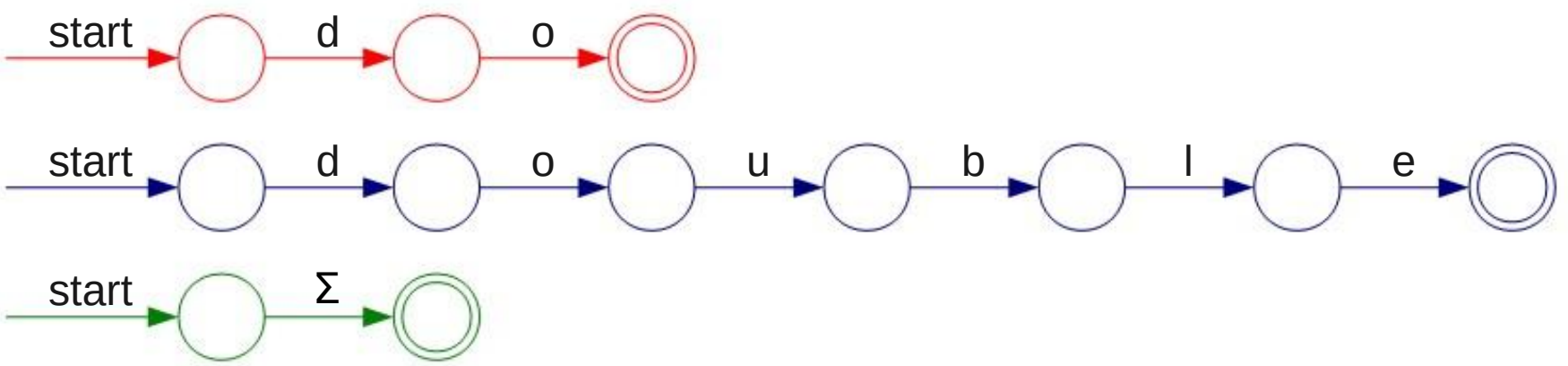
```
T_Do          do
T_Double      double
T_Mystery     [A-Za-z]
```

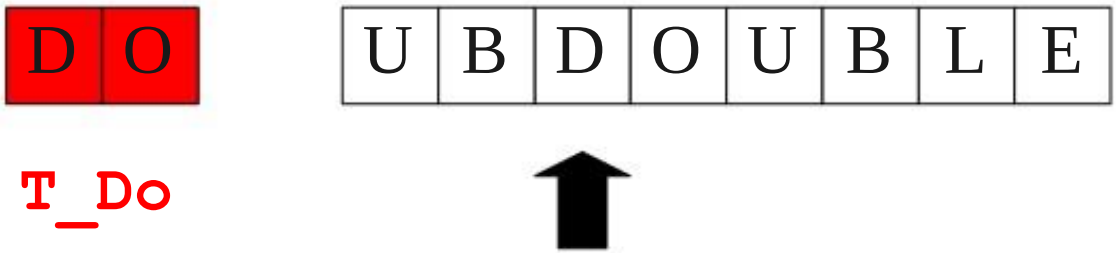




Implementing Maximal Munch

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

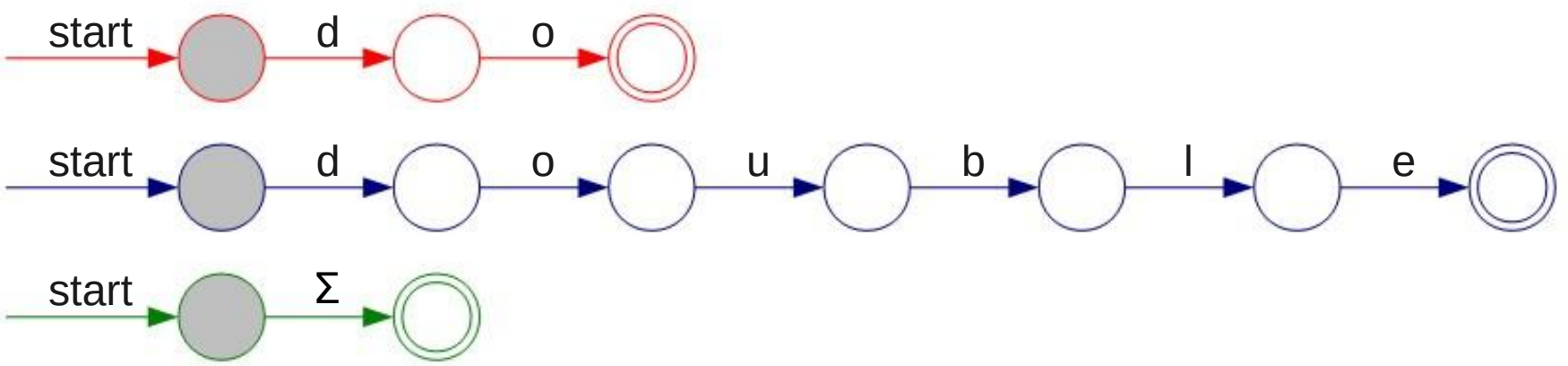


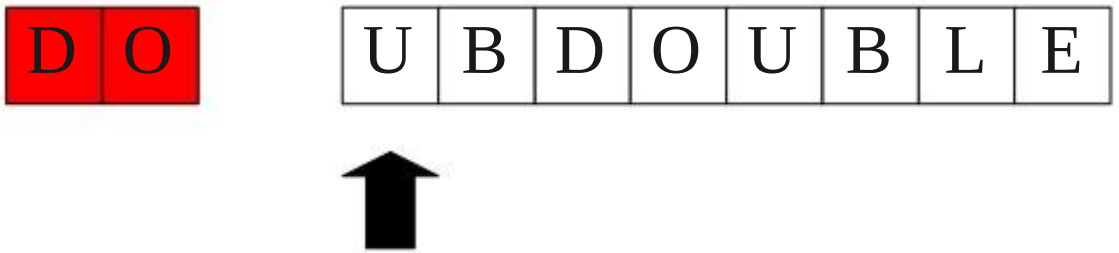


T_Do

Implementing Maximal Munch

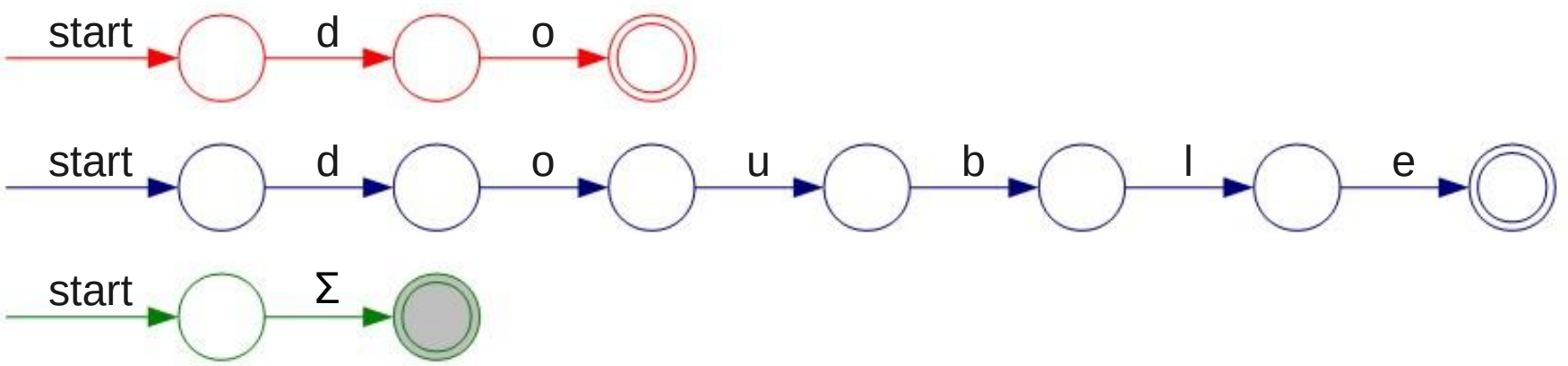
```
T_Do          do
T_Double      double
T_Mystery     [A-Za-z]
```

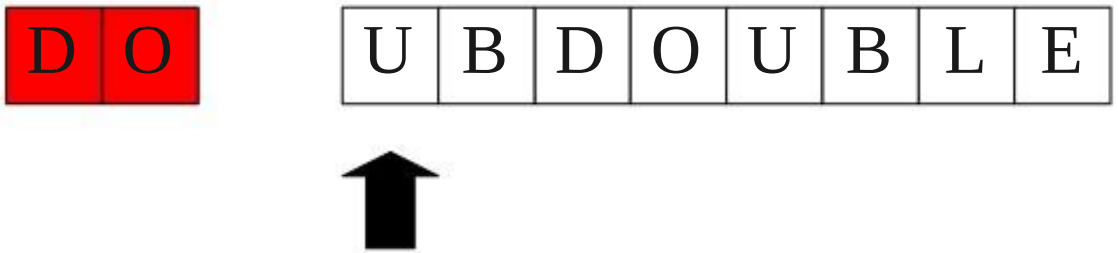




Implementing Maximal Munch

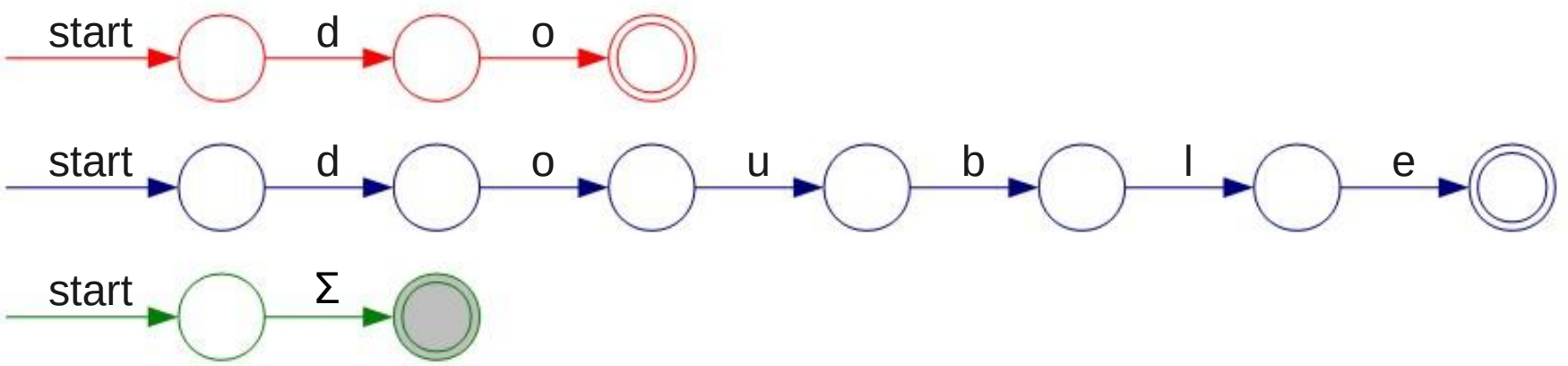
```
T_Do          do
T_Double      double
T_Mystery     [A-Za-z]
```

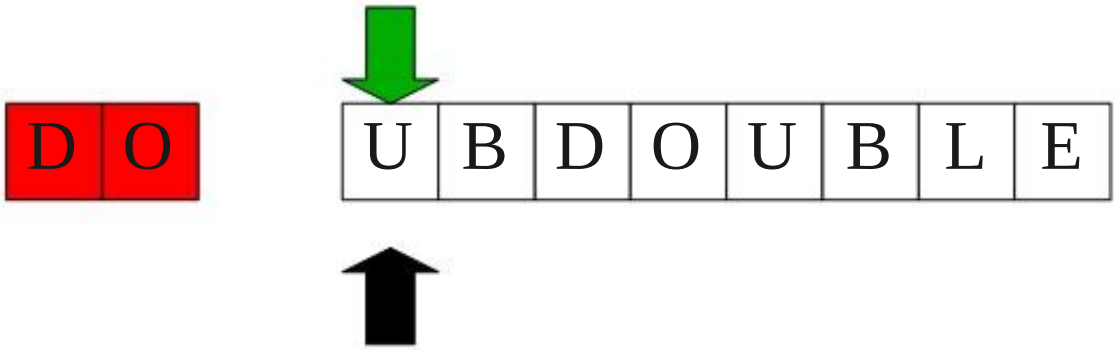




Implementing Maximal Munch

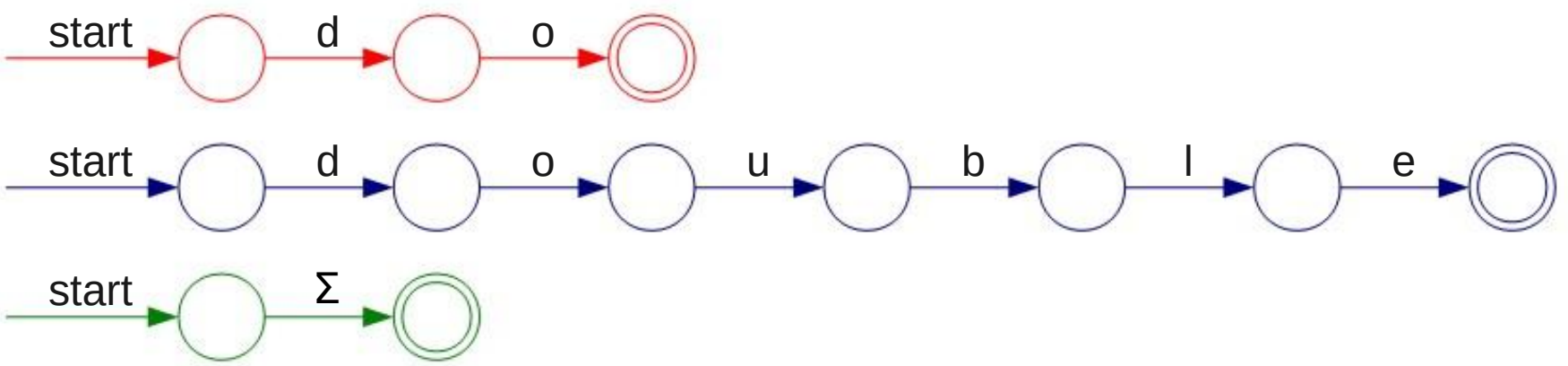
```
T_Do          do
T_Double      double
T_Mystery     [A-Za-z]
```

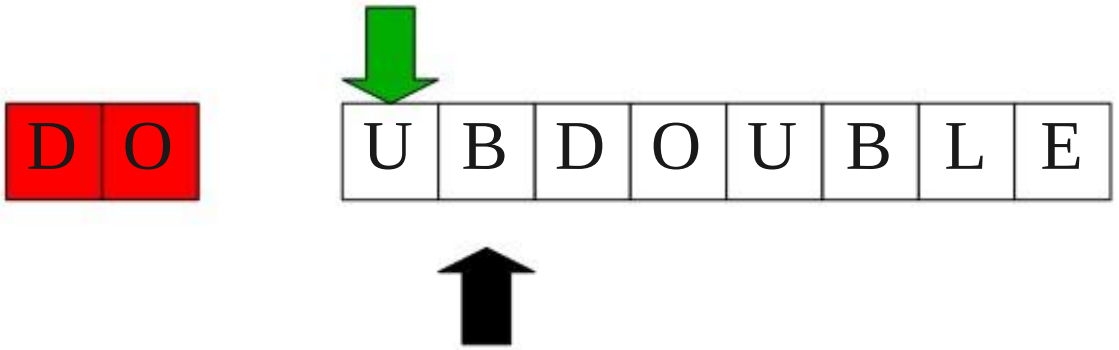




Implementing Maximal Munch

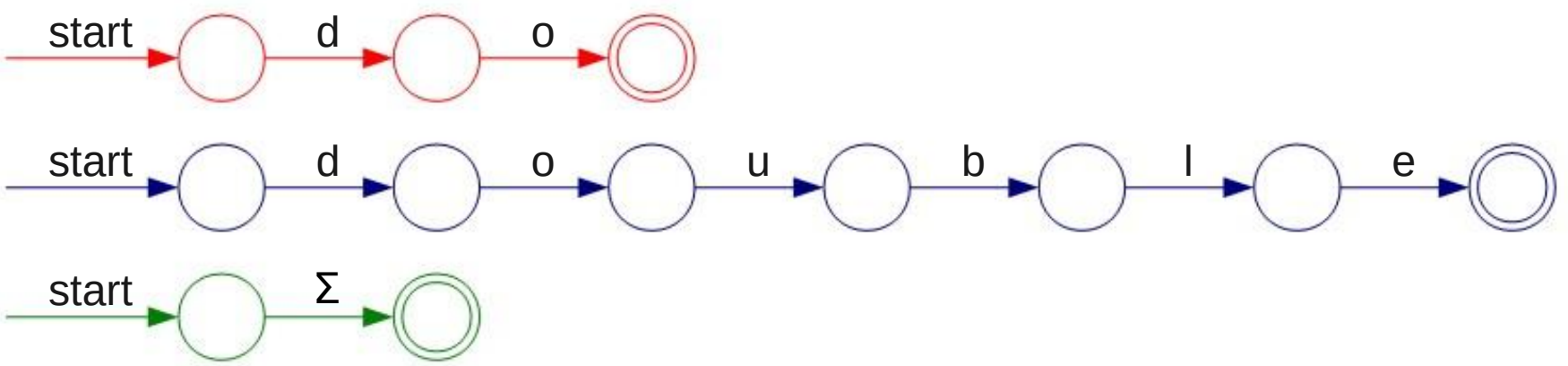
```
T_Do          do
T_Double      double
T_Mystery     [A-Za-z]
```

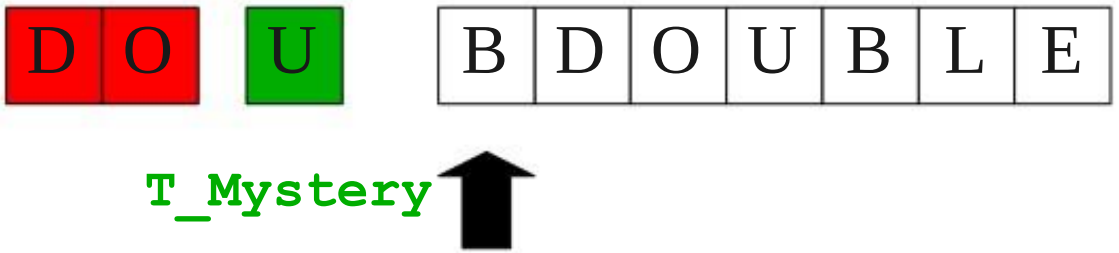




Implementing Maximal Munch

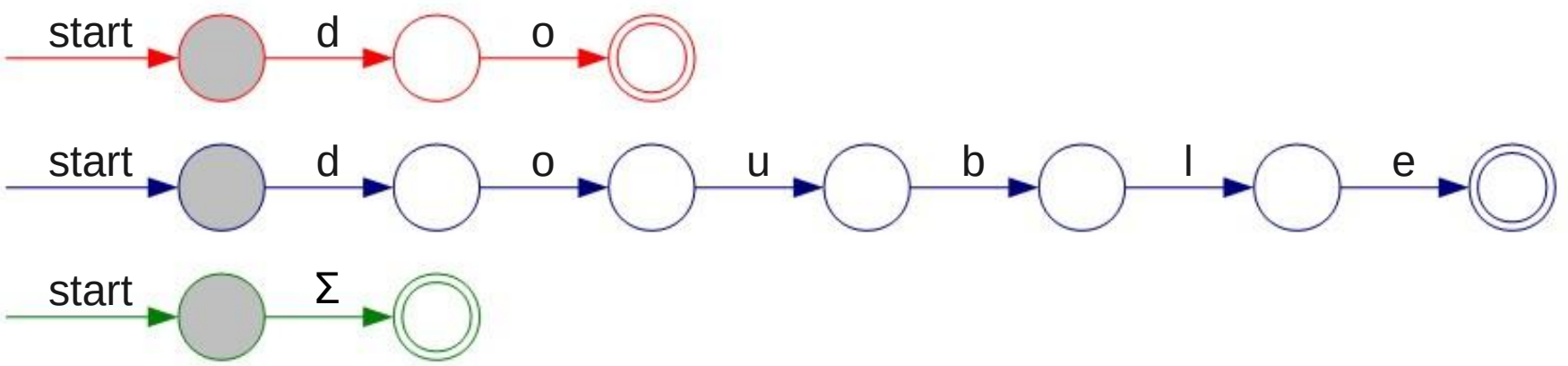
```
T_Do          do
T_Double      double
T_Mystery     [A-Za-z]
```

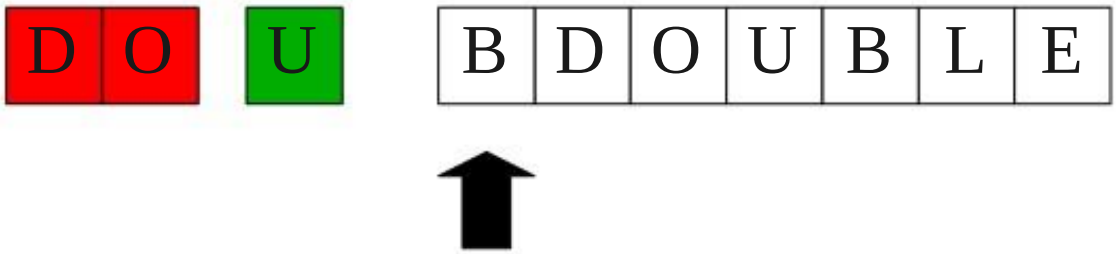




Implementing Maximal Munch

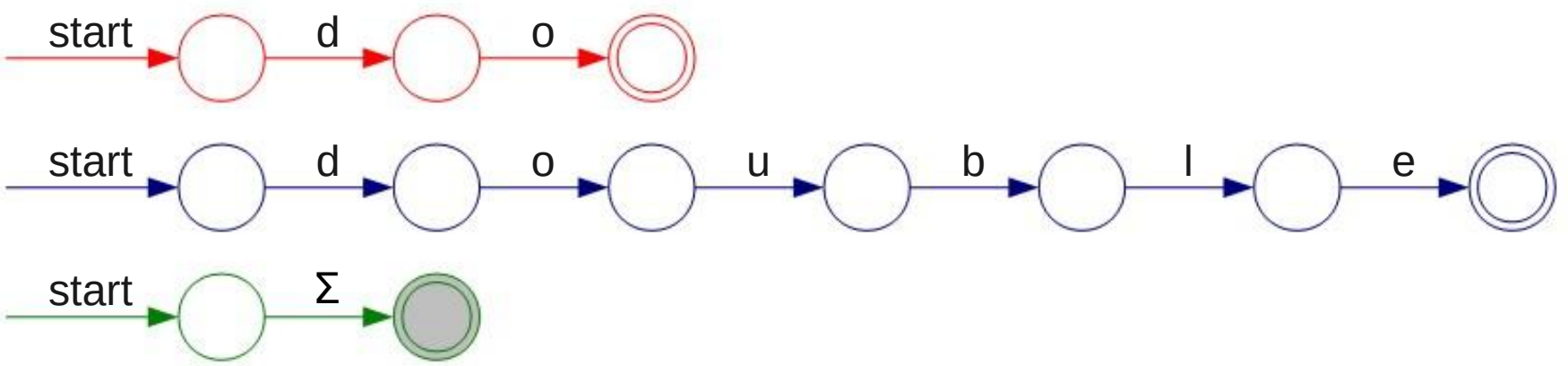
```
T_Do          do
T_Double      double
T_Mystery     [A-Za-z]
```

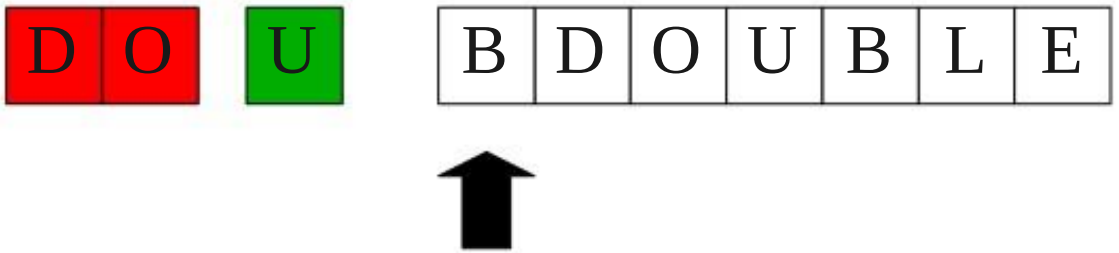




Implementing Maximal Munch

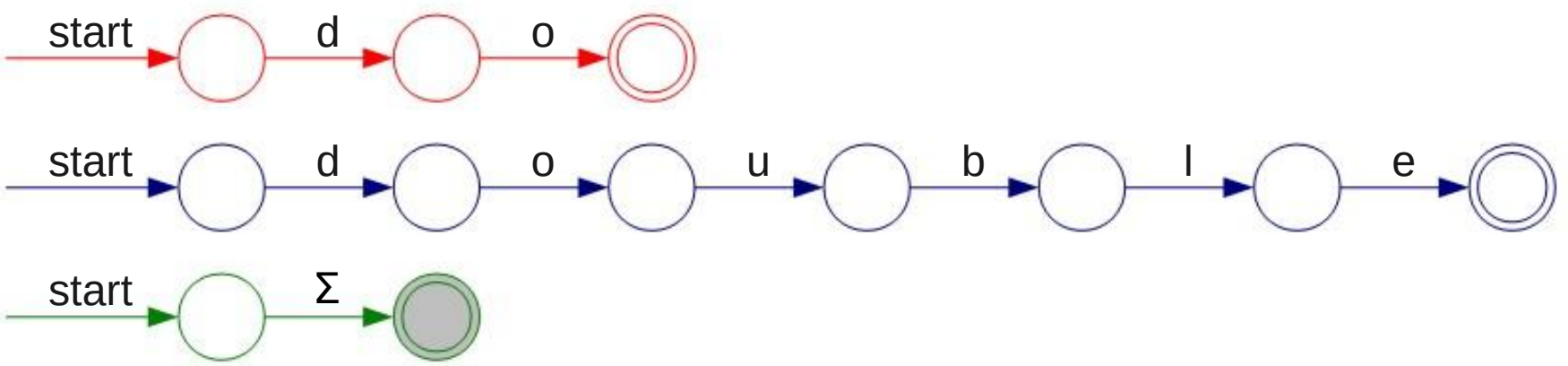
```
T_Do          do
T_Double      double
T_Mystery     [A-Za-z]
```

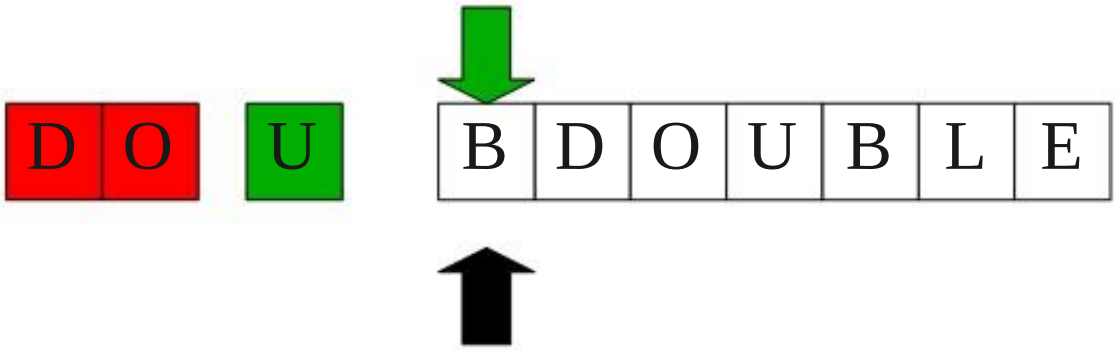




Implementing Maximal Munch

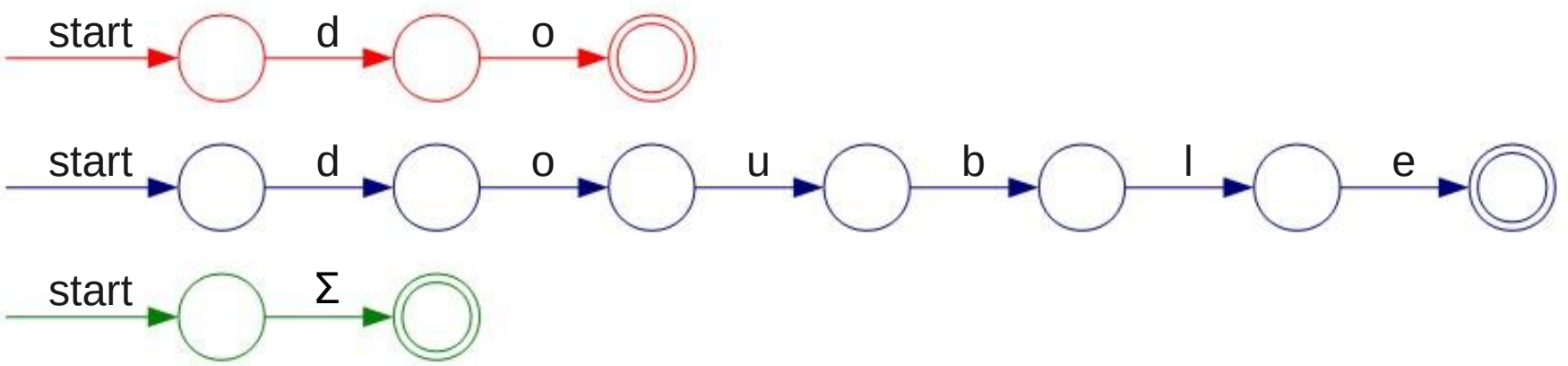
```
T_Do          do
T_Double      double
T_Mystery     [A-Za-z]
```

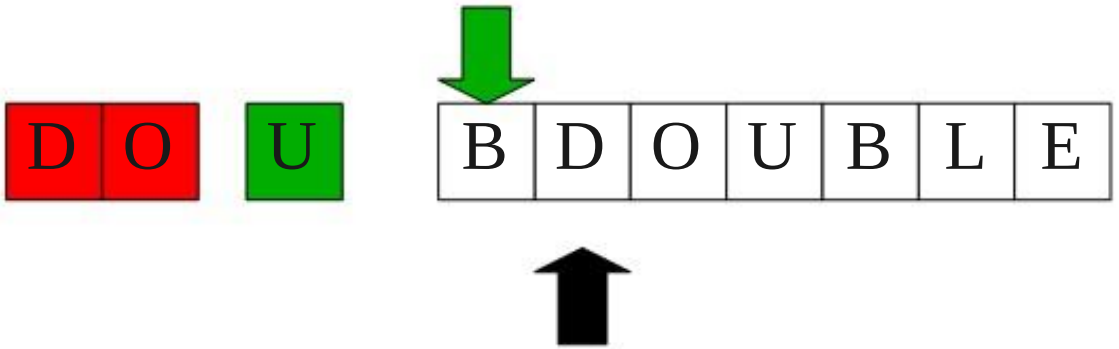




Implementing Maximal Munch

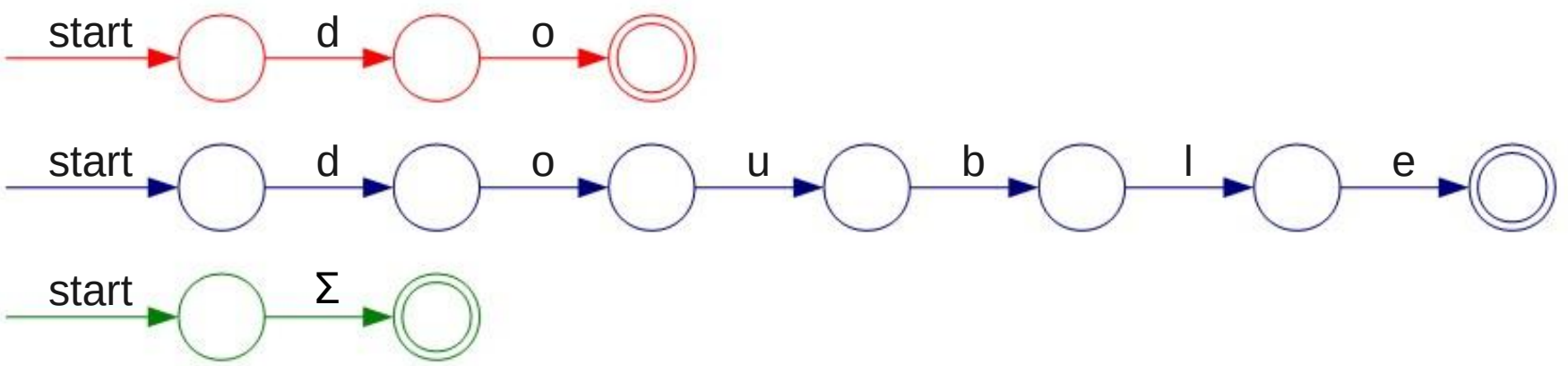
```
T_Do          do
T_Double      double
T_Mystery     [A-Za-z]
```

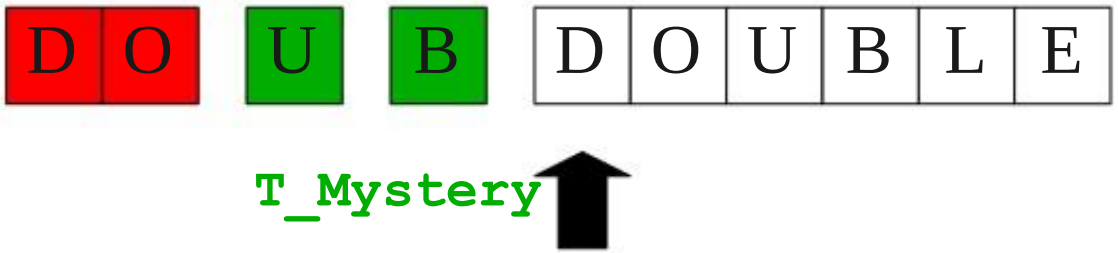




Implementing Maximal Munch

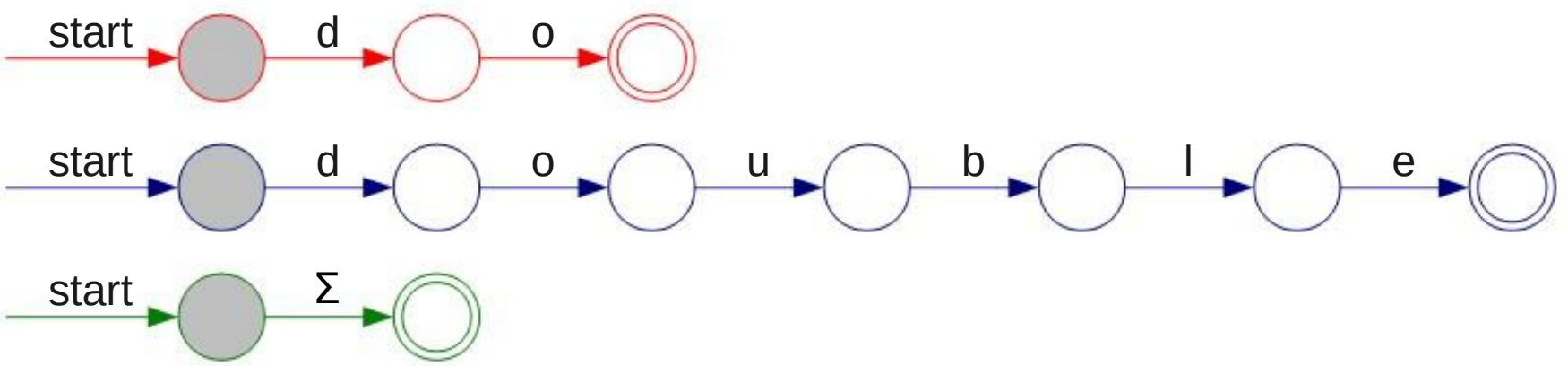
```
T_Do          do
T_Double      double
T_Mystery     [A-Za-z]
```

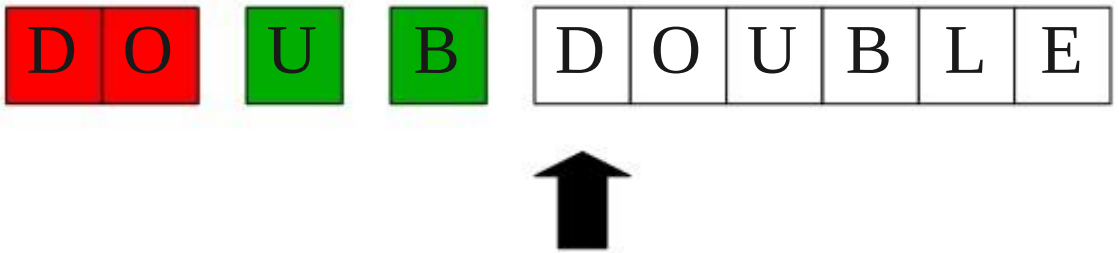




Implementing Maximal Munch

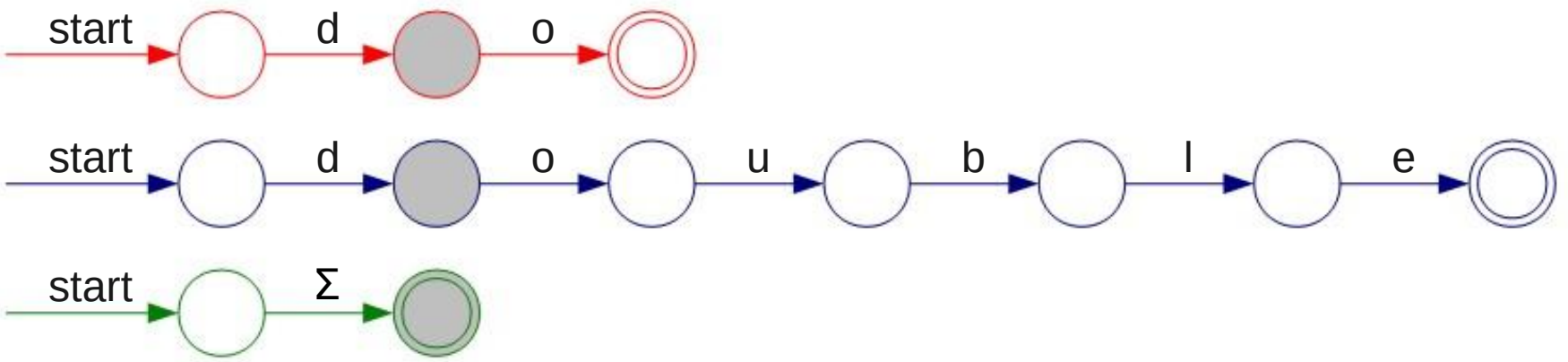
```
T_Do          do
T_Double      double
T_Mystery     [A-Za-z]
```

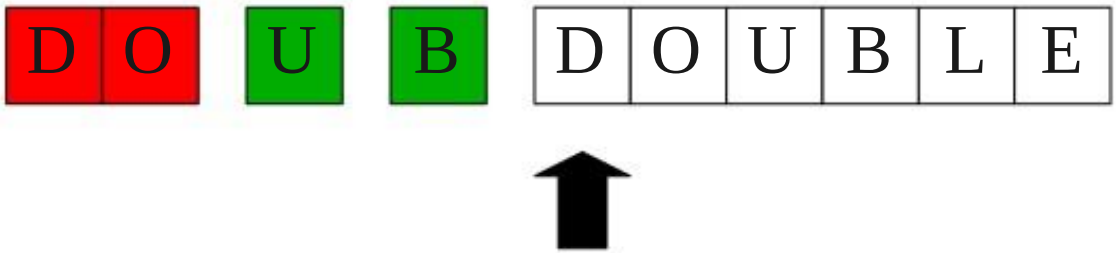




Implementing Maximal Munch

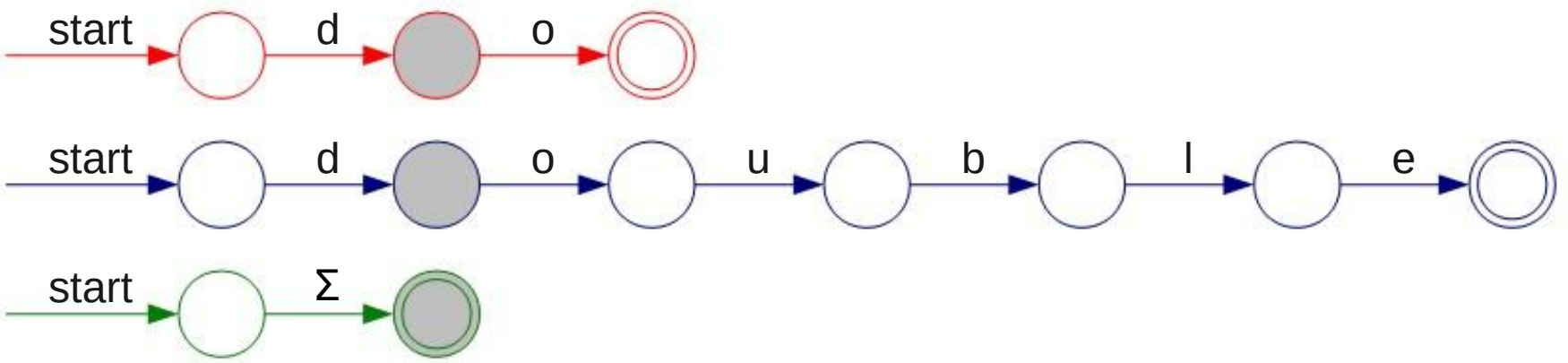
```
T_Do          do
T_Double      double
T_Mystery     [A-Za-z]
```

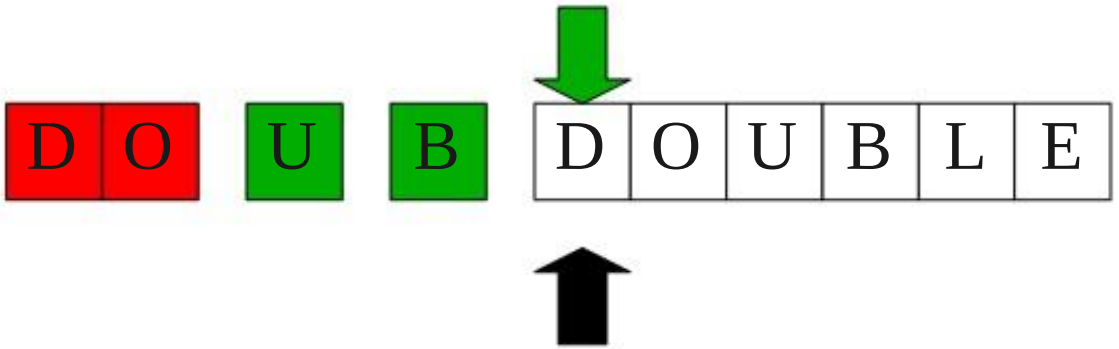




Implementing Maximal Munch

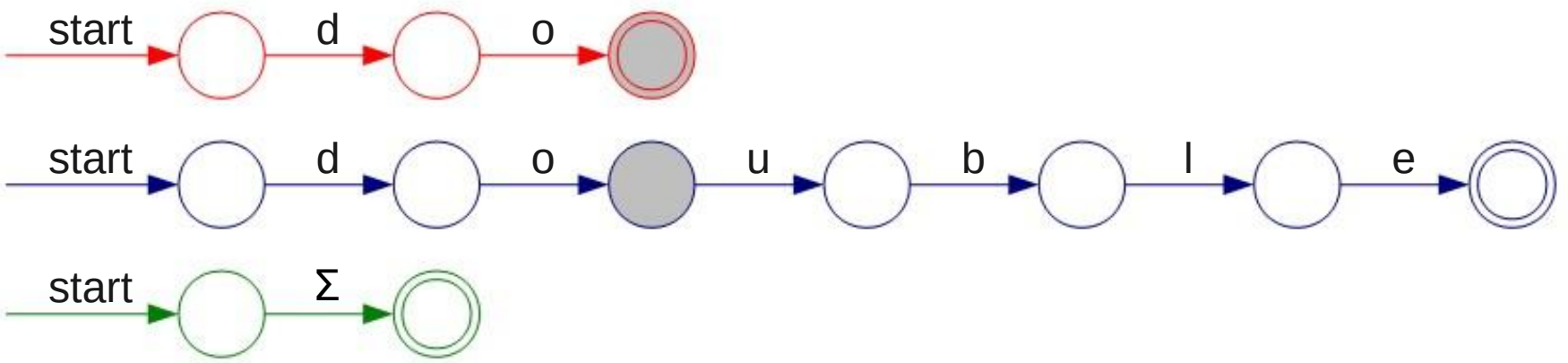
```
T_Do          do
T_Double      double
T_Mystery     [A-Za-z]
```

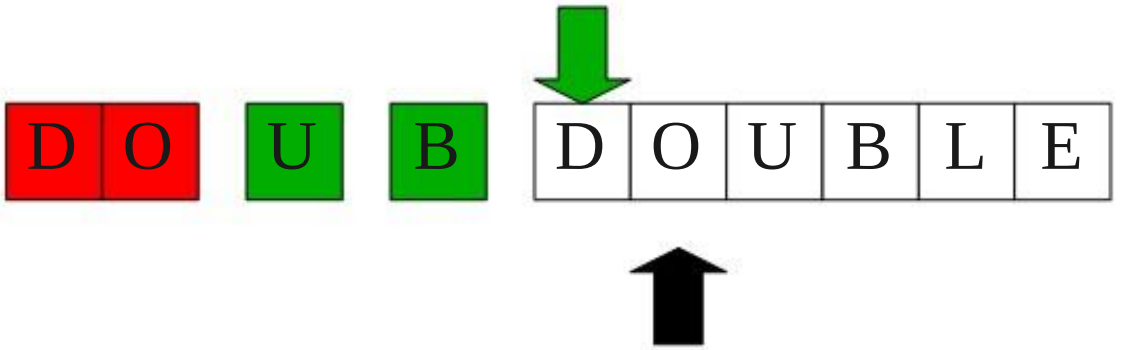




Implementing Maximal Munch

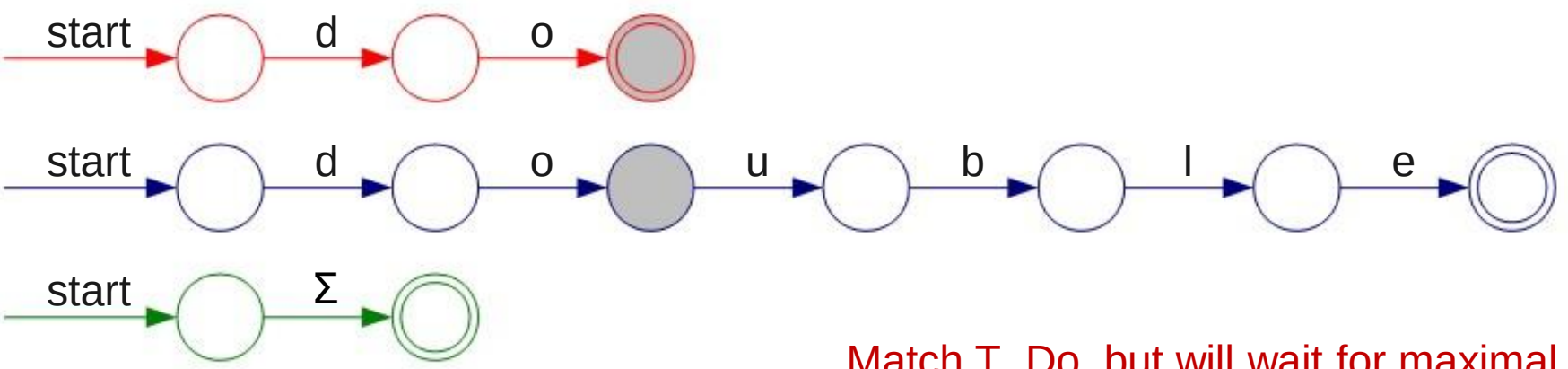
```
T_Do          do
T_Double      double
T_Mystery     [A-Za-z]
```



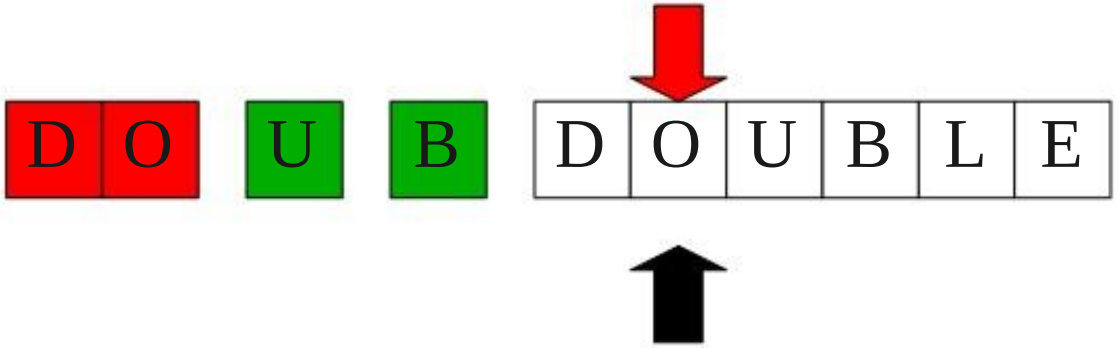


Implementing Maximal Munch

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

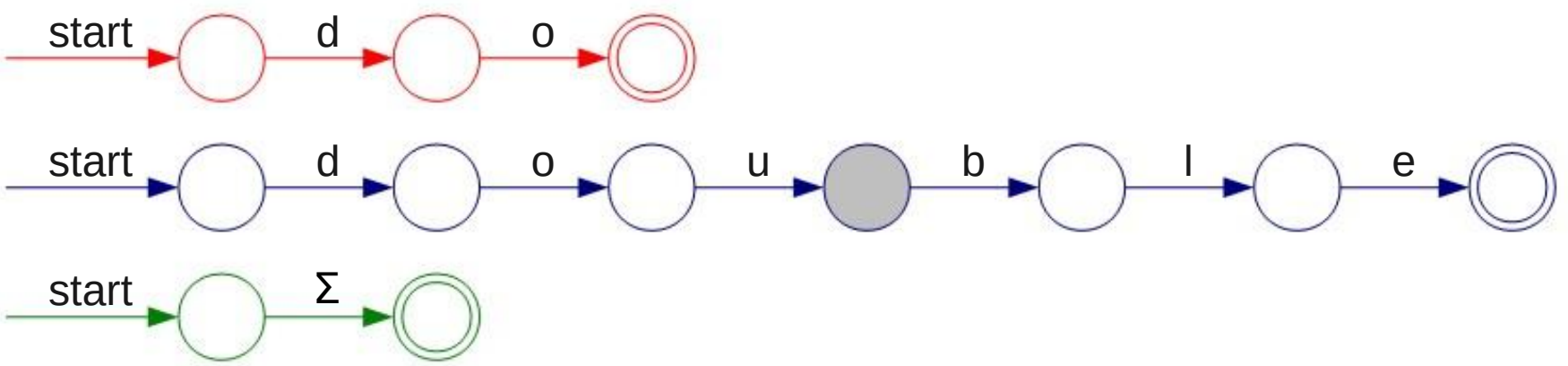


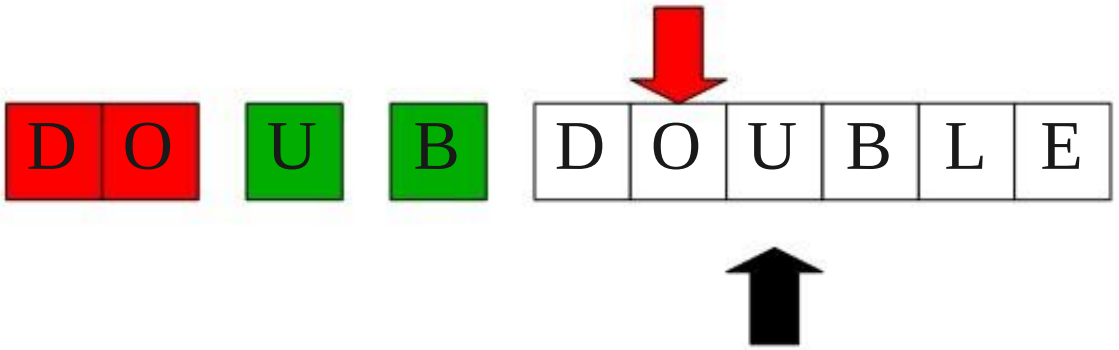
Match T_Do, but will wait for maximal munch



Implementing Maximal Munch

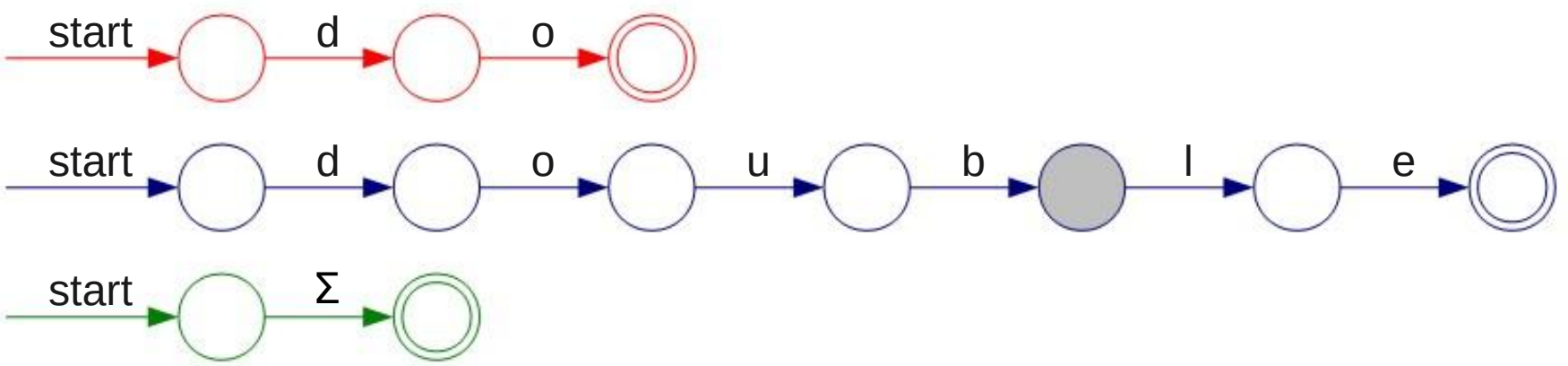
```
T_Do          do
T_Double      double
T_Mystery     [A-Za-z]
```

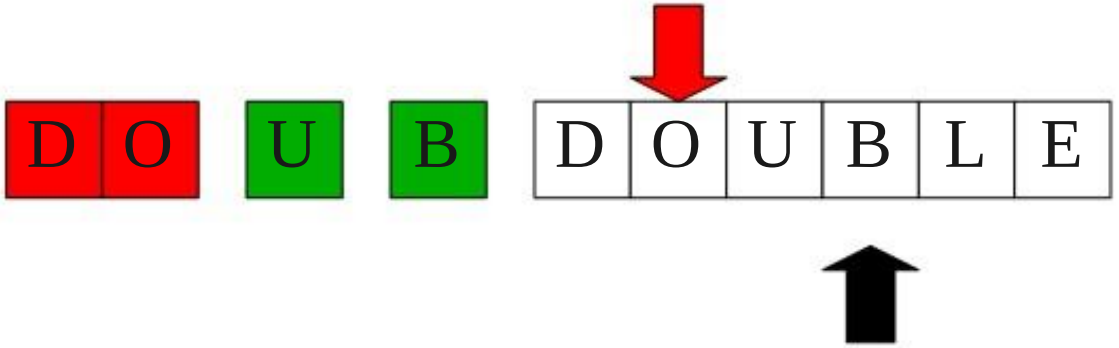




Implementing Maximal Munch

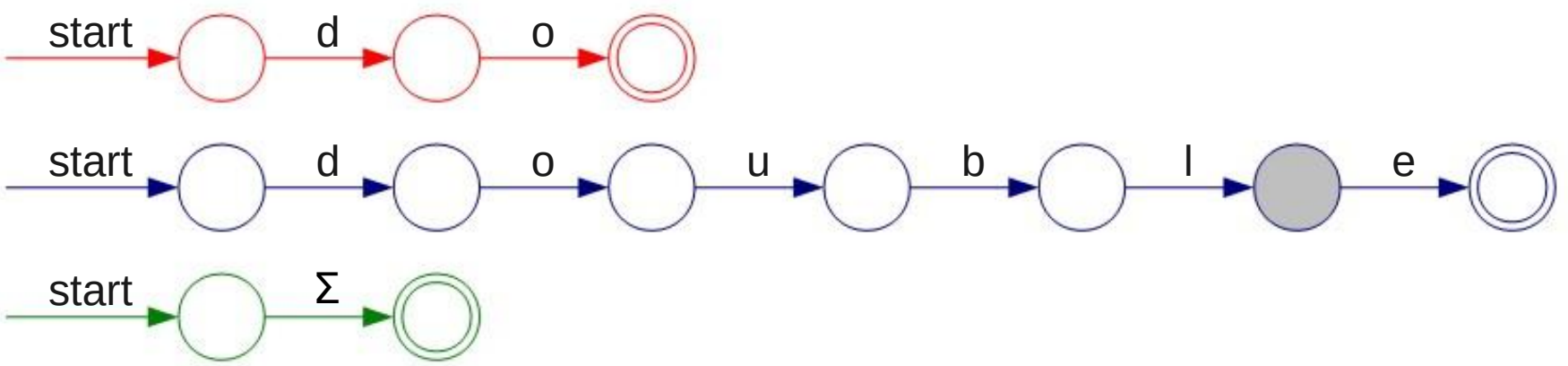
```
T_Do          do
T_Double      double
T_Mystery     [A-Za-z]
```

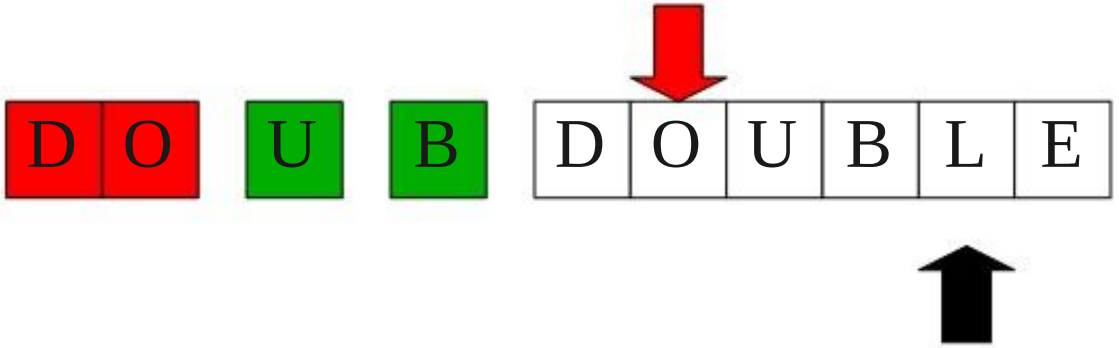




Implementing Maximal Munch

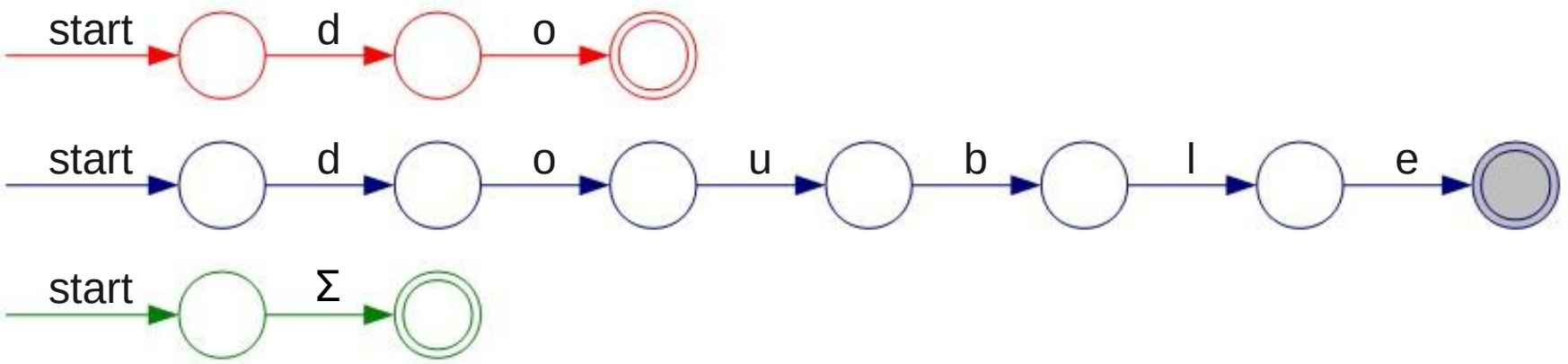
```
T_Do          do
T_Double      double
T_Mystery     [A-Za-z]
```

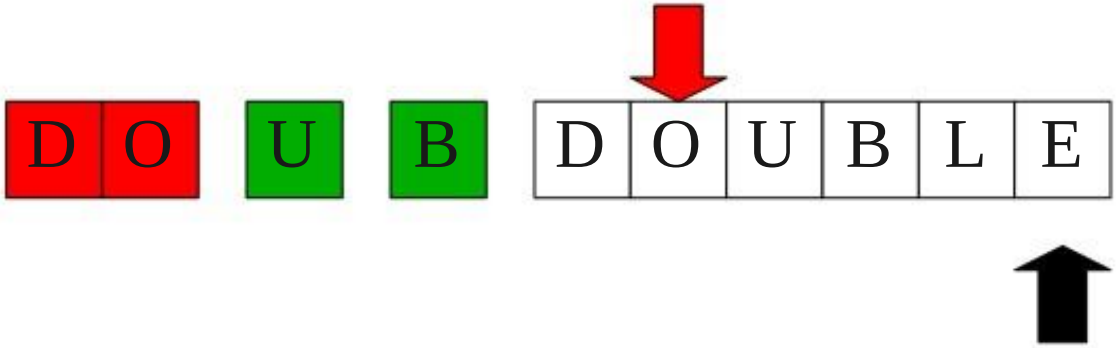




Implementing Maximal Munch

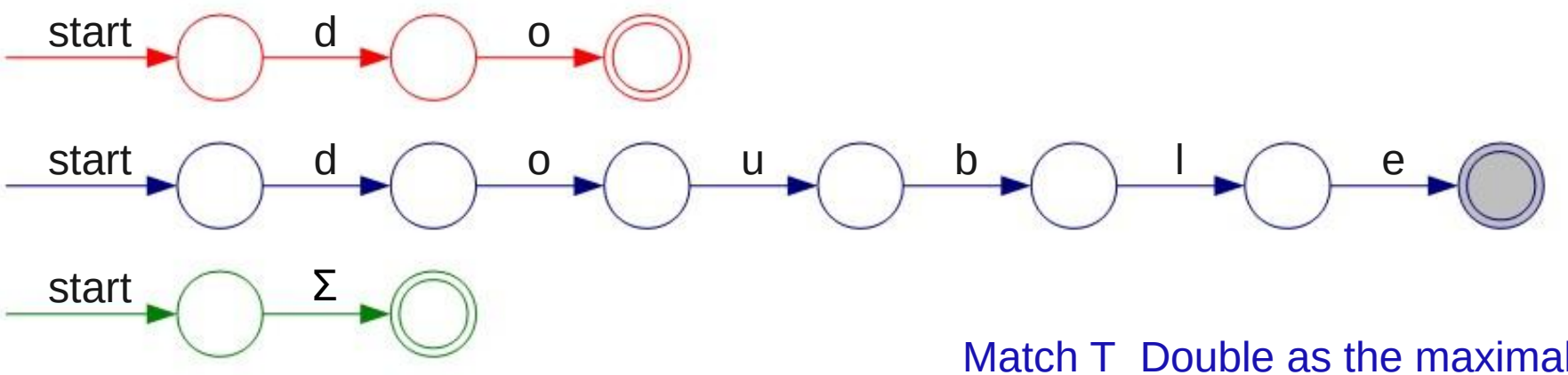
```
T_Do          do
T_Double      double
T_Mystery     [A-Za-z]
```



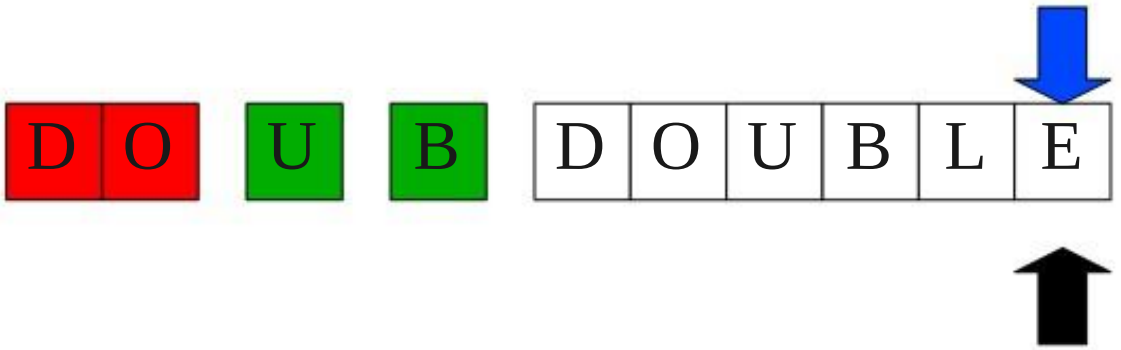


Implementing Maximal Munch

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

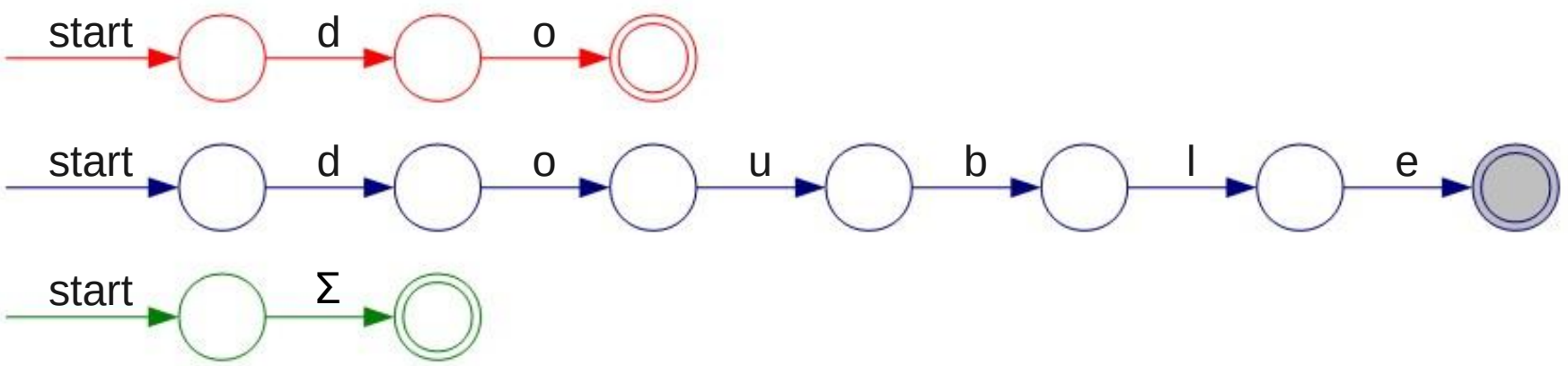


Match T_Double as the maximal munch



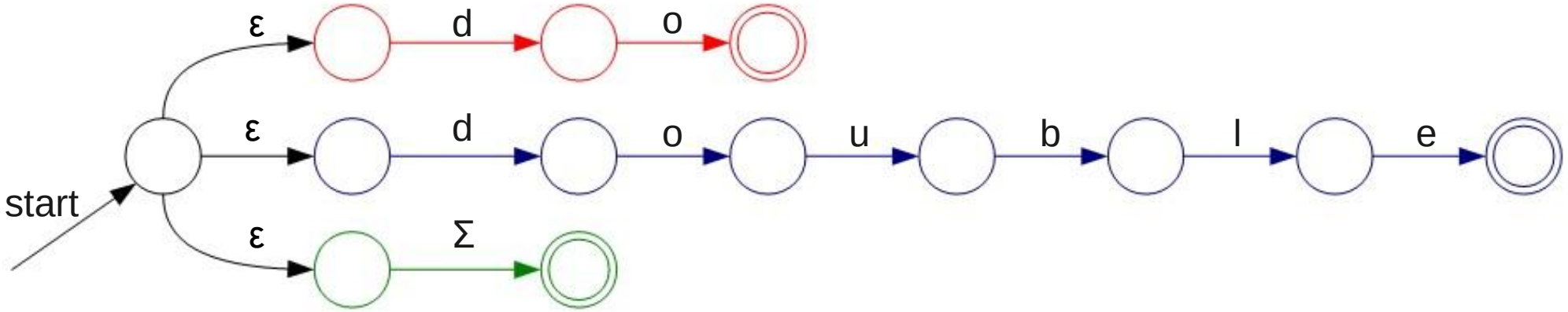
Implementing Maximal Munch

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



T_Double

A Minor Simplification



Conflict

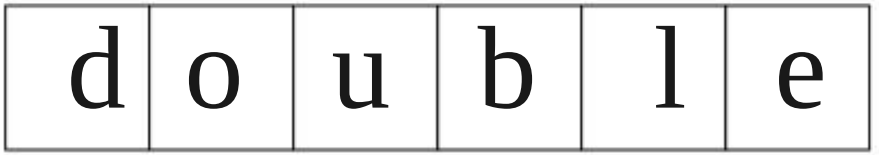
```
T_Do           do
T_Double       double
T_Identifier  [A-Za-z_][A-Za-z0-9_]*
```

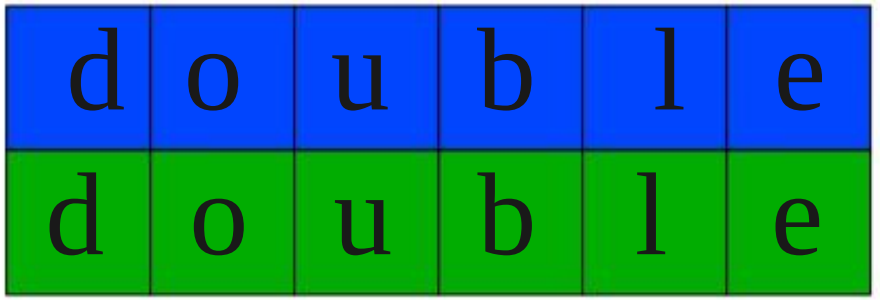
Multiple rules can be executed for the same lexeme. As a result, one lexeme can be recognized as more than one token.

Conflict

```
T_Do            do
T_Double        double
T_Identifier    [A-Za-z_][A-Za-z0-9_]*
```

Example





T_Double

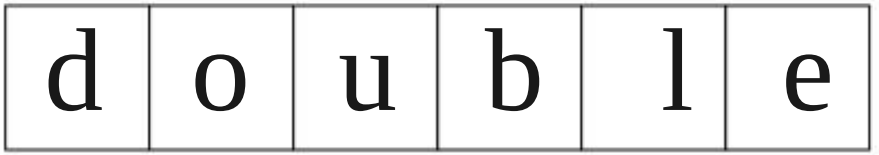
T_Identifier

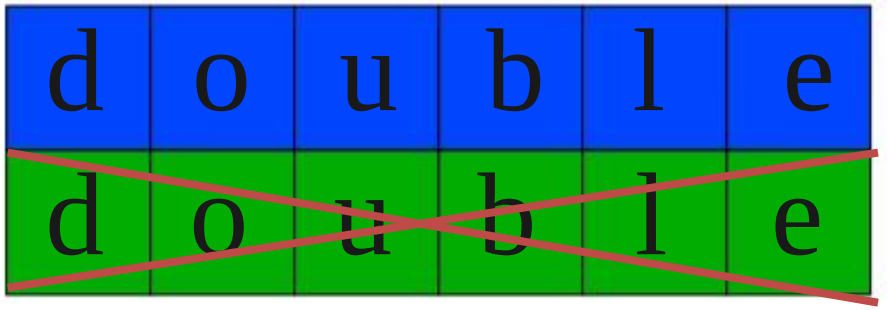
More Tiebreaking

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

Conflict

```
T_Do            do
T_Double        double
T_Identifier    [A-Za-z_][A-Za-z0-9_]*
```





One Last Detail...

- What if **nothing** matches?
- Trick: Add a “catch-all” rule that matches any character and reports an error. Don’t forget to put it at the last line (lowest priority).

Summary of Conflict Resolution

- Construct an automaton for each regular expression.
- Merge them into one automaton by adding a new start state.
- Scan the input, keeping track of the last known match.
- Break ties by choosing higher-precedence matches.
- Have a catch-all rule to handle errors.

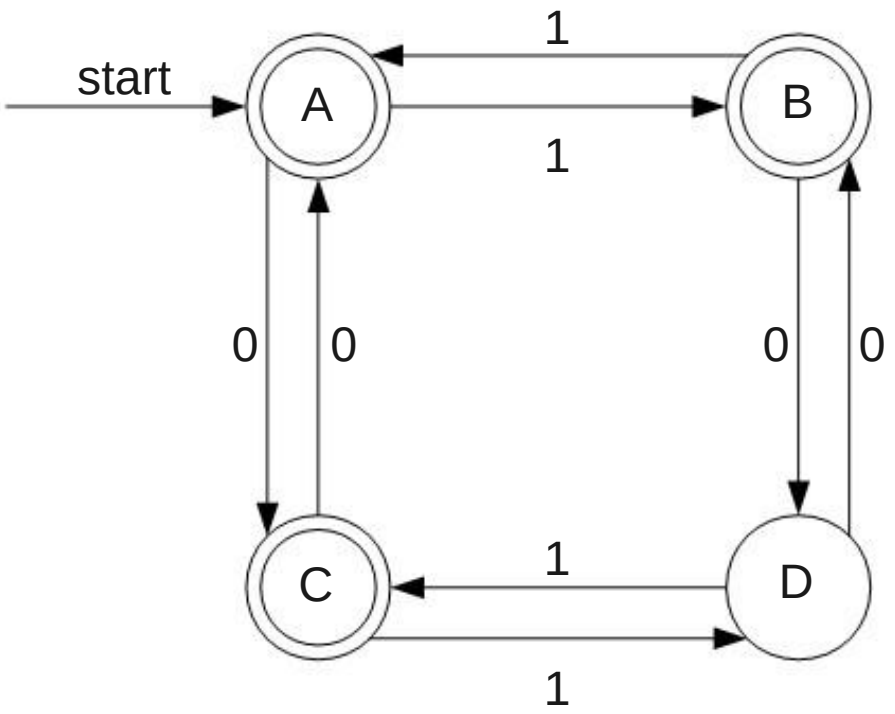
Speeding up the Scanner

DFAs (Review)

- The automata we've seen so far have all been NFAs.

- A **DFA** is like an NFA, but with tighter restrictions:
  - Every state must have **exactly one** transition defined for every letter.
  - ε-moves are not allowed.

A Sample DFA

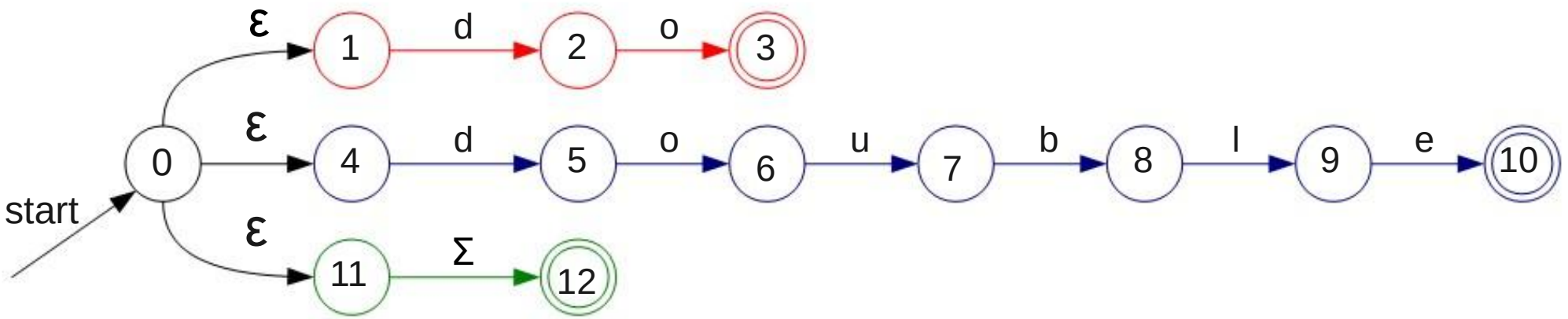


	0	1
A	C	B
B	D	A
C	A	D
D	B	C

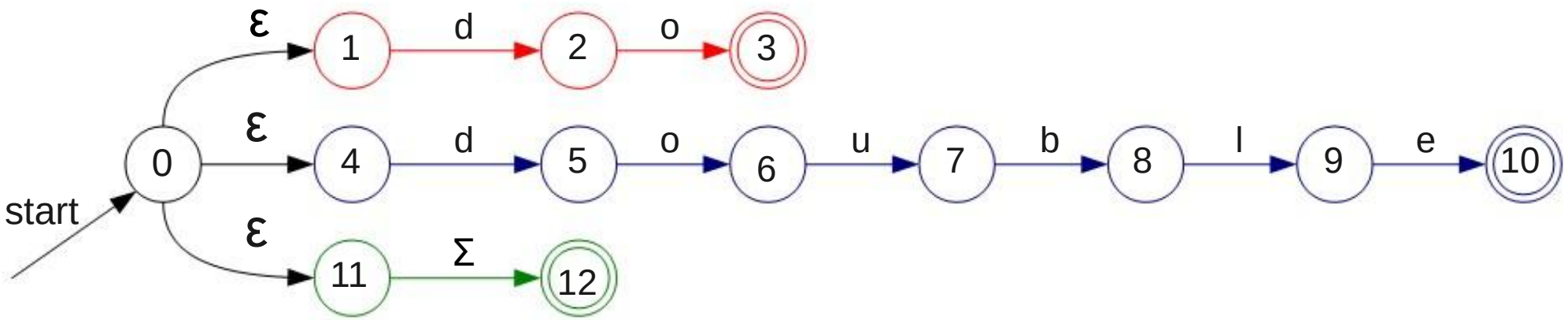
Speeding up Matching

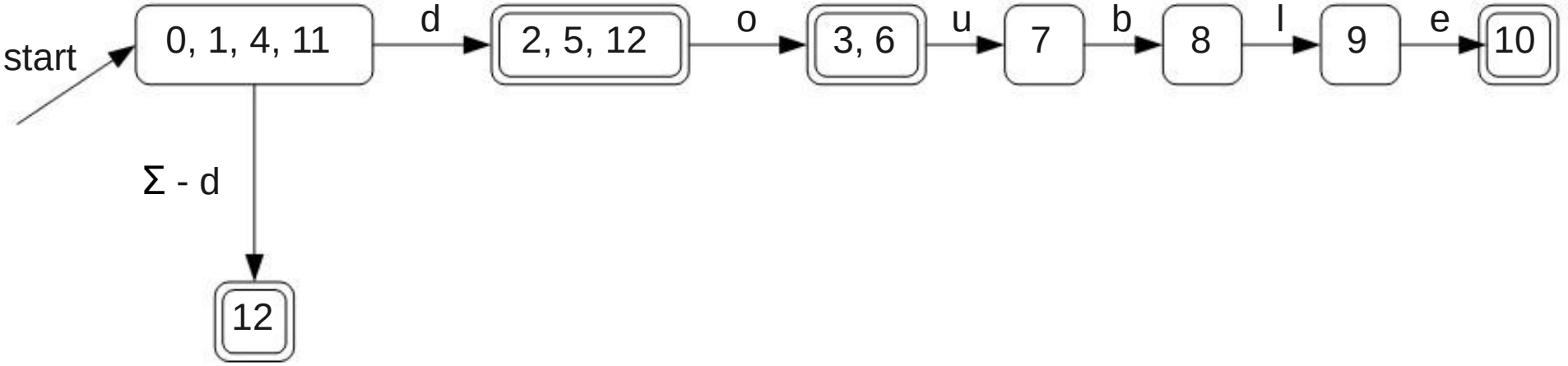
- DFAs usually take shorter execution time than NFAs.

From NFA to DFA

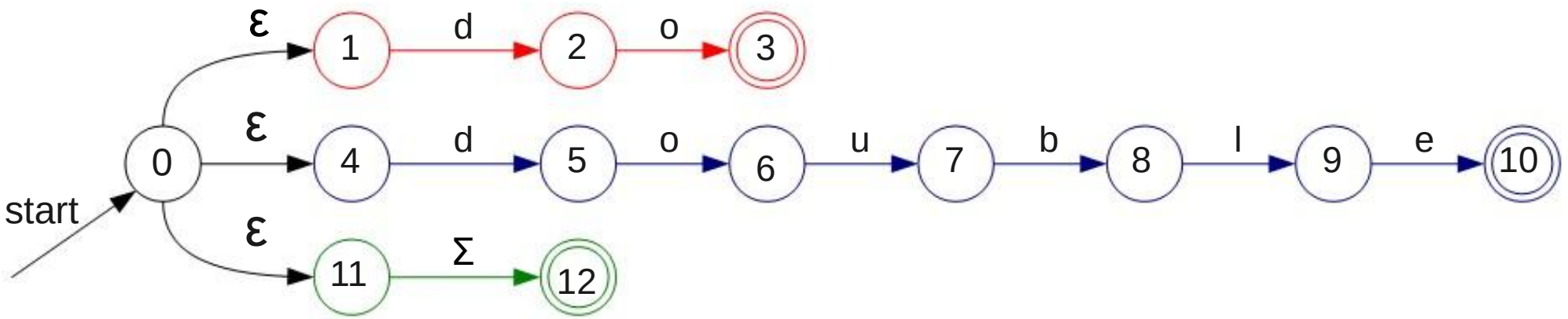


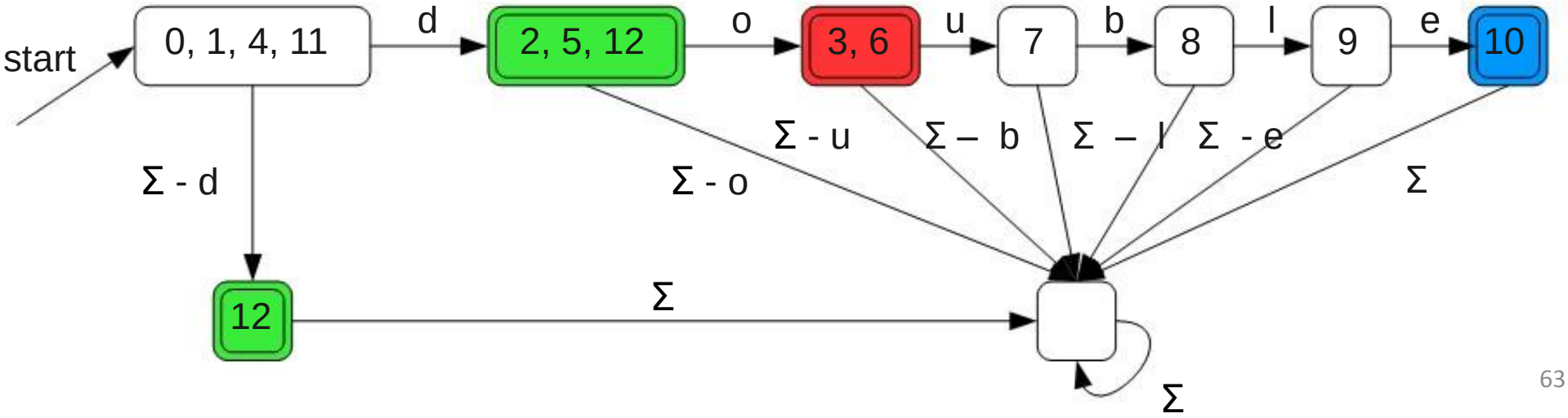
From NFA to DFA





From NFA to DFA





Introduction to `flex`

(Lexical Analysis Tool)

What is `flex`?

- Automated tool for generating scanner (lexer).
- Uses maximal-munch/precedence system.
- Internally, builds a DFA from regular expressions.
- Plus several more features...

A Simple **flex** File

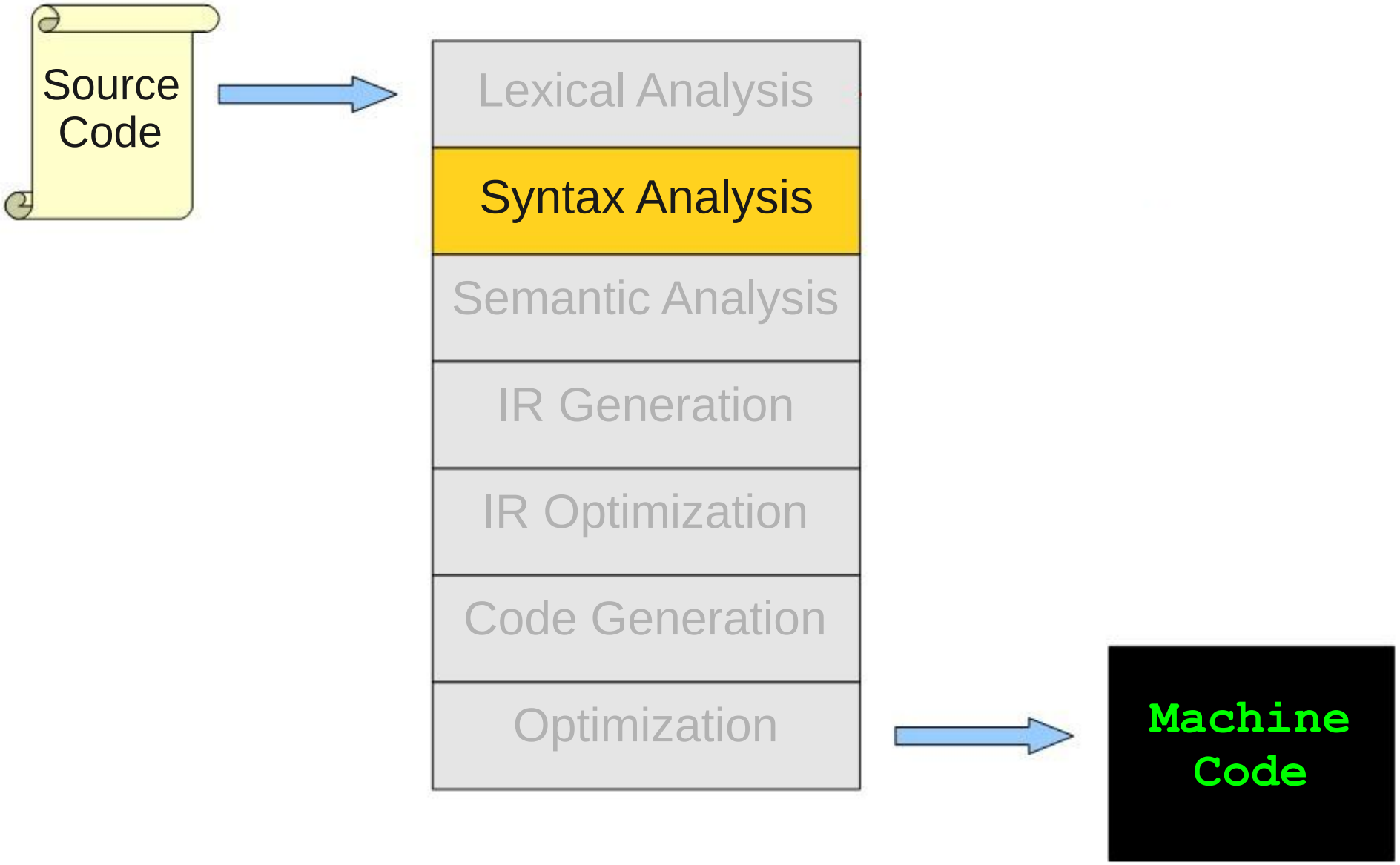
```
%%
[A-Za-z]*          printf("Word\n");
[0-9]*             printf("Number\n");
[ \t\n]            ;
.                  printf("Undefine Symbol\n");
```

Further reading : http://flex.sourceforge.net/manual/

Summary

- Lexical analysis splits input text into **tokens** holding a **lexeme** and an **attribute**.
- Lexemes are sets of strings often defined with **regular expressions**.
- Regular expressions can be converted to **NFAs** and from there to **DFAs**.
- **Maximal-munch** using an automaton allows for fast scanning.
- Not all tokens come directly from the source code.

Next Time


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