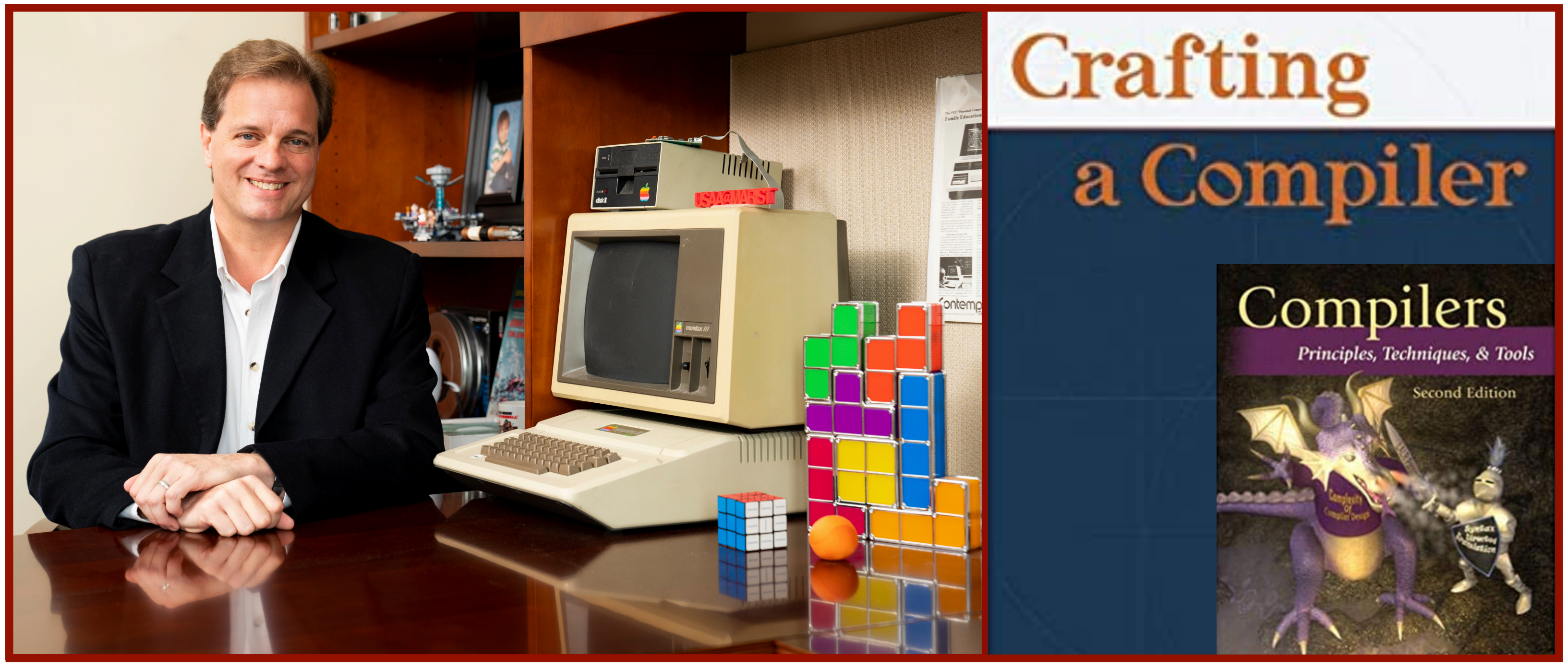
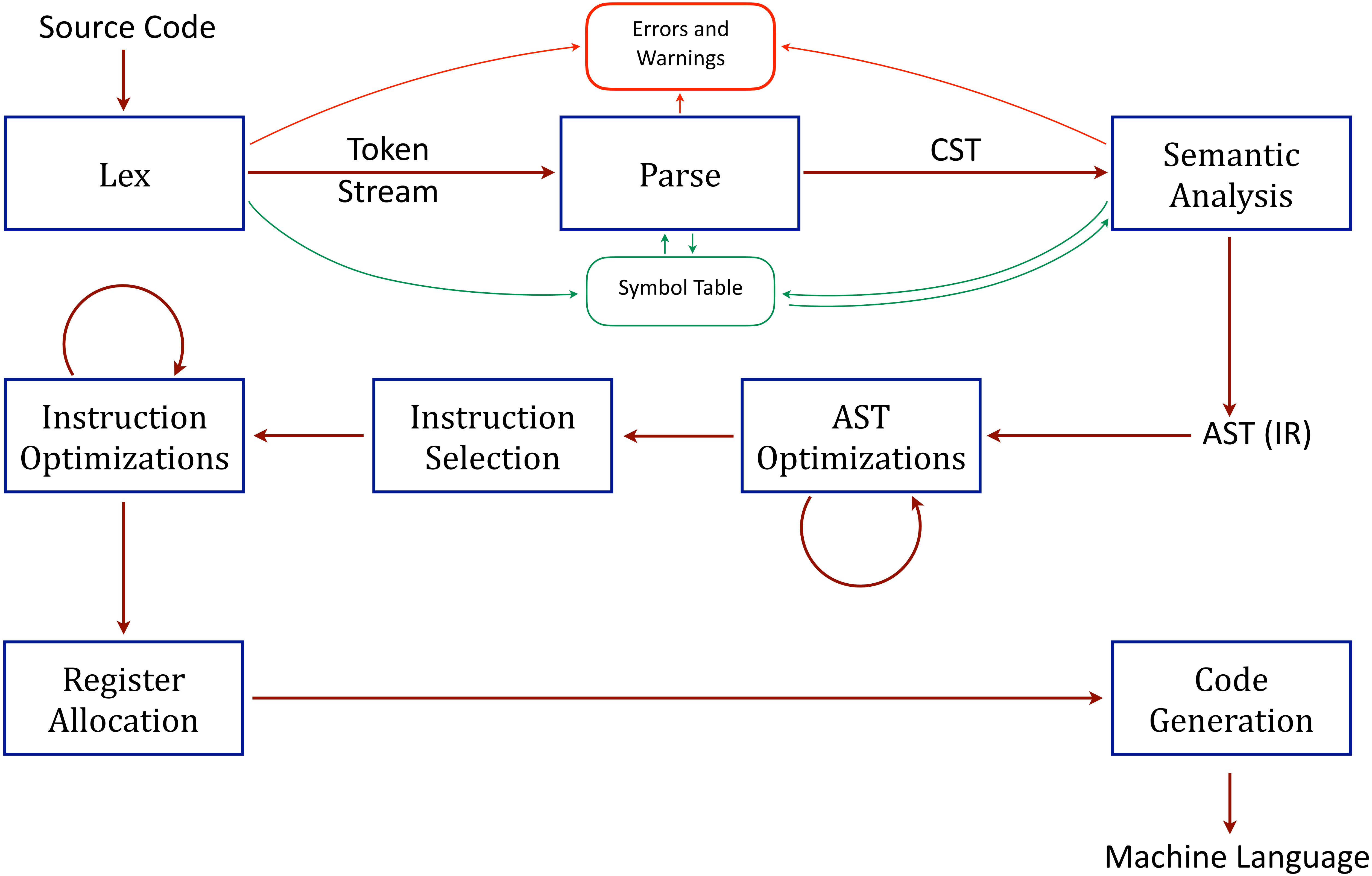
Lexical Analysis

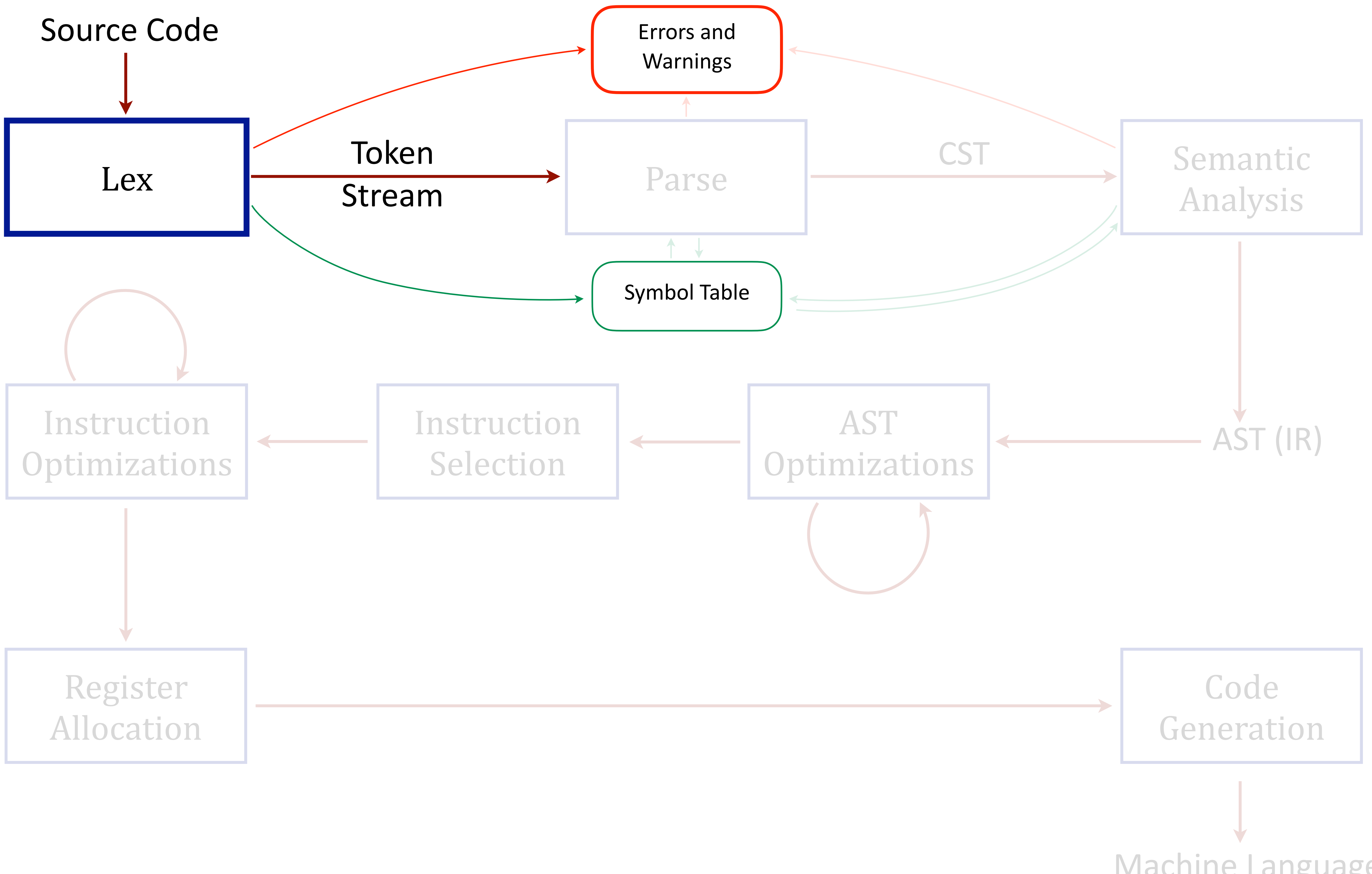


Alan G. Labouseur, Ph.D.
Alan.Labouseur@Marist.edu

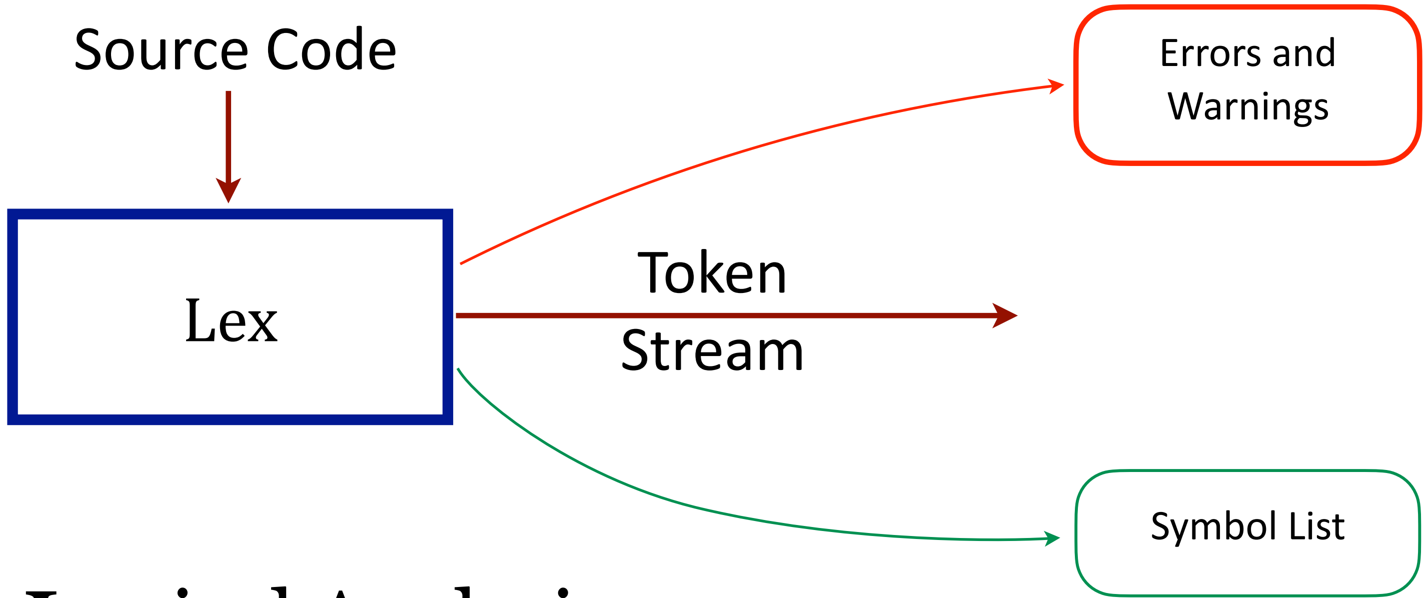
Compiler — High Level View



Compiler — High Level View



Lexical Analysis



Lexical Analysis

- Maps characters into an ordered stream of tokens

  `x := x + y`

  becomes

  `<id,x> <assign> <id,x> <add> <id,y>`
- Typical tokens: `id int while print if`
- Eliminates white space and comments
- Reports meaningful errors and warnings
- Produces a token stream for Parse.

- Focus on words/lexemes/tokens

Lexical Analysis

Only concerned with the words/syntax.

Not concerned with structure or meaning (sentences, type, scope).

Uses white space to help determine boundaries, then discards it.

Keeps track of line number for every token.

Builds the initial symbol list.

Definitions:

- A **token** is a sequence of characters that we'll treat as a unit in the grammar of our language. (https://www.labouseur.com/courses/compilers/grammar.pdf)
- A **pattern** is a description of the form legal tokens can take.
- A **lexeme** is the sequence of characters in the source code that match a pattern for a token in the language.

Choosing Good Tokens

- Dependent on language.
- Varies by language.

- Typically . . .
  - keywords (different from identifiers)
  - identifiers (different from keywords)
  - punctuation symbols
  - digits
  - individual characters

  get their own tokens.

- Discard white space and comments, but only after making use of them if possible.

Choosing Good Tokens

Why is this important? What if we didn't choose good tokens?

- PL/1 had no reserved words at first . . .

```
if then then then = else; else else = then;
```

. . . so keywords could be used as identifiers (variable and other names). Turns out that wasn't great.

Choosing Good Tokens

Why is this important? What if we didn't choose good tokens?

- PL/1 had no reserved words at first

```
if then then then = else; else else = then;
```

- FORTRAN and Algol 68 ignored spaces, even for token identification

```
do 10 i = 1,25

do 10 i = 1.25
```

Choosing Good Tokens

Why is this important? What if we didn't choose good tokens?

- PL/1 had no reserved words

```
if then then then = else; else else = then;
```

- FORTRAN and Algol 68 ignored spaces, even for token identification

`do 10 i = 1,25`	Loop from 1 to 25. (`10` is a label.)
`do 10 i = 1.25`	Assignment. (`do10i` is a variable.)

Lexical Analysis

```
/* Test case for WhileStatement.
   Prints 23458 */
{
   int a
   a = 1
   {
      while (a != 5) {
         a = 1 + a
         print(a)
      }
      print(3 + a)
   }
}$
```

Lexical Analysis takes **nicely formatted** source code

- newlines
- indenting
- spaces between expressions

and turns it into an ordered stream of tokens.

```
INFO  Compilation started
INFO  Compiling Program 1
DEBUG - Lexer - LBrace [ { ] found at (2:0)
DEBUG - Lexer - Type [ int ] found at (3:1)
DEBUG - Lexer - Id [ a ] found at (3:7)
DEBUG - Lexer - Id [ a ] found at (4:3)
DEBUG - Lexer - Assign [ = ] found at (4:5)
DEBUG - Lexer - Digit [ 1 ] found at (4:7)
DEBUG - Lexer - LBrace [ { ] found at (5:3)
DEBUG - Lexer - While [ while ] found at (6:4)
DEBUG - Lexer - LParen [ ( ] found at (6:12)
DEBUG - Lexer - Id [ a ] found at (6:13)
DEBUG - Lexer - BoolOp [ != ] found at (6:15)
DEBUG - Lexer - Digit [ 5 ] found at (6:17)
DEBUG - Lexer - RParen [ ) ] found at (6:18)
DEBUG - Lexer - LBrace [ { ] found at (6:20)
DEBUG - Lexer - Id [ a ] found at (7:10)
DEBUG - Lexer - Assign [ = ] found at (7:12)
DEBUG - Lexer - Digit [ 1 ] found at (7:14)
DEBUG - Lexer - IntOp [ + ] found at (7:16)
DEBUG - Lexer - Id [ a ] found at (7:18)
DEBUG - Lexer - Print [ print ] found at (8:8)
DEBUG - Lexer - LParen [ ( ] found at (8:15)
DEBUG - Lexer - Id [ a ] found at (8:16)
DEBUG - Lexer - RParen [ ) ] found at (8:17)
DEBUG - Lexer - RBrace [ } ] found at (9:6)
DEBUG - Lexer - Print [ print ] found at (10:4)
DEBUG - Lexer - LParen [ ( ] found at (10:11)
DEBUG - Lexer - Digit [ 3 ] found at (10:12)
DEBUG - Lexer - IntOp [ + ] found at (10:14)
DEBUG - Lexer - Id [ a ] found at (10:16)
DEBUG - Lexer - RParen [ ) ] found at (10:17)
DEBUG - Lexer - RBrace [ } ] found at (11:3)
DEBUG - Lexer - RBrace [ } ] found at (12:0)
DEBUG - Lexer - EOP [ $ ] found at (12:2)
INFO  Lexical Analysis complete with 0 WARNING(S) and 0 ERROR(S)
```

Lexical Analysis

```
/* Test case for WhileStatement.
Prints 23458 */{int a a=1 {while(a!
=5){a=1+a print(a)} print(3+a)}}$
```

Lexical Analysis takes **barely formatted** source code

- all one line
- no indenting
- few spaces between expressions

and turns it into an ordered stream of tokens.

```
INFO  Compilation started
INFO  Compiling Program 1
DEBUG - Lexer - LBrace [ { ] found at (1:47)
DEBUG - Lexer - Type [ int ] found at (1:46)
DEBUG - Lexer - Id [ a ] found at (1:52)
DEBUG - Lexer - Id [ a ] found at (1:54)
DEBUG - Lexer - Assign [ = ] found at (1:55)
DEBUG - Lexer - Digit [ 1 ] found at (1:56)
DEBUG - Lexer - LBrace [ { ] found at (1:58)
DEBUG - Lexer - While [ while ] found at (1:57)
DEBUG - Lexer - LParen [ ( ] found at (1:64)
DEBUG - Lexer - Id [ a ] found at (1:65)
DEBUG - Lexer - BoolOp [ != ] found at (1:66)
DEBUG - Lexer - Digit [ 5 ] found at (1:67)
DEBUG - Lexer - RParen [ ) ] found at (1:68)
DEBUG - Lexer - LBrace [ { ] found at (1:69)
DEBUG - Lexer - Id [ a ] found at (1:70)
DEBUG - Lexer - Assign [ = ] found at (1:71)
DEBUG - Lexer - Digit [ 1 ] found at (1:72)
DEBUG - Lexer - IntOp [ + ] found at (1:73)
DEBUG - Lexer - Id [ a ] found at (1:74)
DEBUG - Lexer - Print [ print ] found at (1:74)
DEBUG - Lexer - LParen [ ( ] found at (1:81)
DEBUG - Lexer - Id [ a ] found at (1:82)
DEBUG - Lexer - RParen [ ) ] found at (1:83)
DEBUG - Lexer - RBrace [ } ] found at (1:84)
DEBUG - Lexer - Print [ print ] found at (1:84)
DEBUG - Lexer - LParen [ ( ] found at (1:91)
DEBUG - Lexer - Digit [ 3 ] found at (1:92)
DEBUG - Lexer - IntOp [ + ] found at (1:93)
DEBUG - Lexer - Id [ a ] found at (1:94)
DEBUG - Lexer - RParen [ ) ] found at (1:95)
DEBUG - Lexer - RBrace [ } ] found at (1:96)
DEBUG - Lexer - RBrace [ } ] found at (1:97)
DEBUG - Lexer - EOP [ $ ] found at (1:98)
INFO  Lexical Analysis complete with 0 WARNING(S) and 0 ERROR(S)
```

Lexical Analysis

```
/* Test case for WhileStatement.
Prints 23458 */{intaa=1{while(a!=5)
{a=1+aprint(a)}print(3+a)}}$
```

Lexical Analysis takes **unformatted** source code
- all one line
- no indenting
- no spaces between expressions

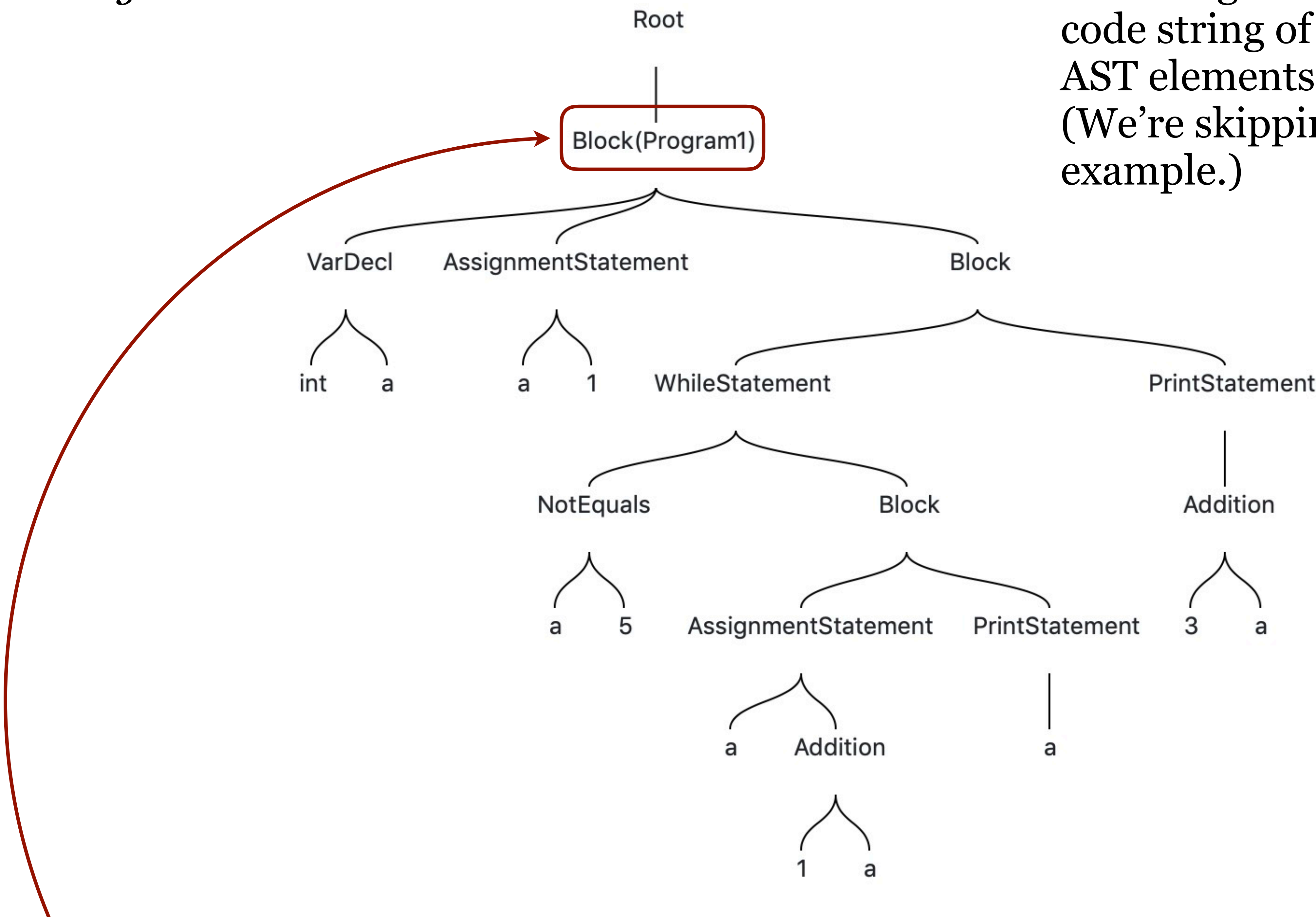
and turns it into an ordered stream of tokens.

```
INFO  Compilation started
INFO  Compiling Program 1
DEBUG - Lexer - LBrace [ { ] found at (1:47)
DEBUG - Lexer - Type [ int ] found at (1:46)
DEBUG - Lexer - Id [ a ] found at (1:51)
DEBUG - Lexer - Id [ a ] found at (1:52)
DEBUG - Lexer - Assign [ = ] found at (1:53)
DEBUG - Lexer - Digit [ 1 ] found at (1:54)
DEBUG - Lexer - LBrace [ { ] found at (1:55)
DEBUG - Lexer - While [ while ] found at (1:54)
DEBUG - Lexer - LParen [ ( ] found at (1:61)
DEBUG - Lexer - Id [ a ] found at (1:62)
DEBUG - Lexer - BoolOp [ != ] found at (1:63)
DEBUG - Lexer - Digit [ 5 ] found at (1:64)
DEBUG - Lexer - RParen [ ) ] found at (1:65)
DEBUG - Lexer - LBrace [ { ] found at (1:66)
DEBUG - Lexer - Id [ a ] found at (1:67)
DEBUG - Lexer - Assign [ = ] found at (1:68)
DEBUG - Lexer - Digit [ 1 ] found at (1:69)
DEBUG - Lexer - IntOp [ + ] found at (1:70)
DEBUG - Lexer - Id [ a ] found at (1:71)
DEBUG - Lexer - Print [ print ] found at (1:70)
DEBUG - Lexer - LParen [ ( ] found at (1:77)
DEBUG - Lexer - Id [ a ] found at (1:78)
DEBUG - Lexer - RParen [ ) ] found at (1:79)
DEBUG - Lexer - RBrace [ } ] found at (1:80)
DEBUG - Lexer - Print [ print ] found at (1:79)
DEBUG - Lexer - LParen [ ( ] found at (1:86)
DEBUG - Lexer - Digit [ 3 ] found at (1:87)
DEBUG - Lexer - IntOp [ + ] found at (1:88)
DEBUG - Lexer - Id [ a ] found at (1:89)
DEBUG - Lexer - RParen [ ) ] found at (1:90)
DEBUG - Lexer - RBrace [ } ] found at (1:91)
DEBUG - Lexer - RBrace [ } ] found at (1:92)
DEBUG - Lexer - EOP [ $ ] found at (1:93)
INFO  Lexical Analysis complete with 0 WARNING(S) and 0 ERROR(S)
```

Lexical Analysis

looking ahead . . .

Goal: to get from the source code string of characters to AST elements and structures. (We're skipping *Parse* in this example.)

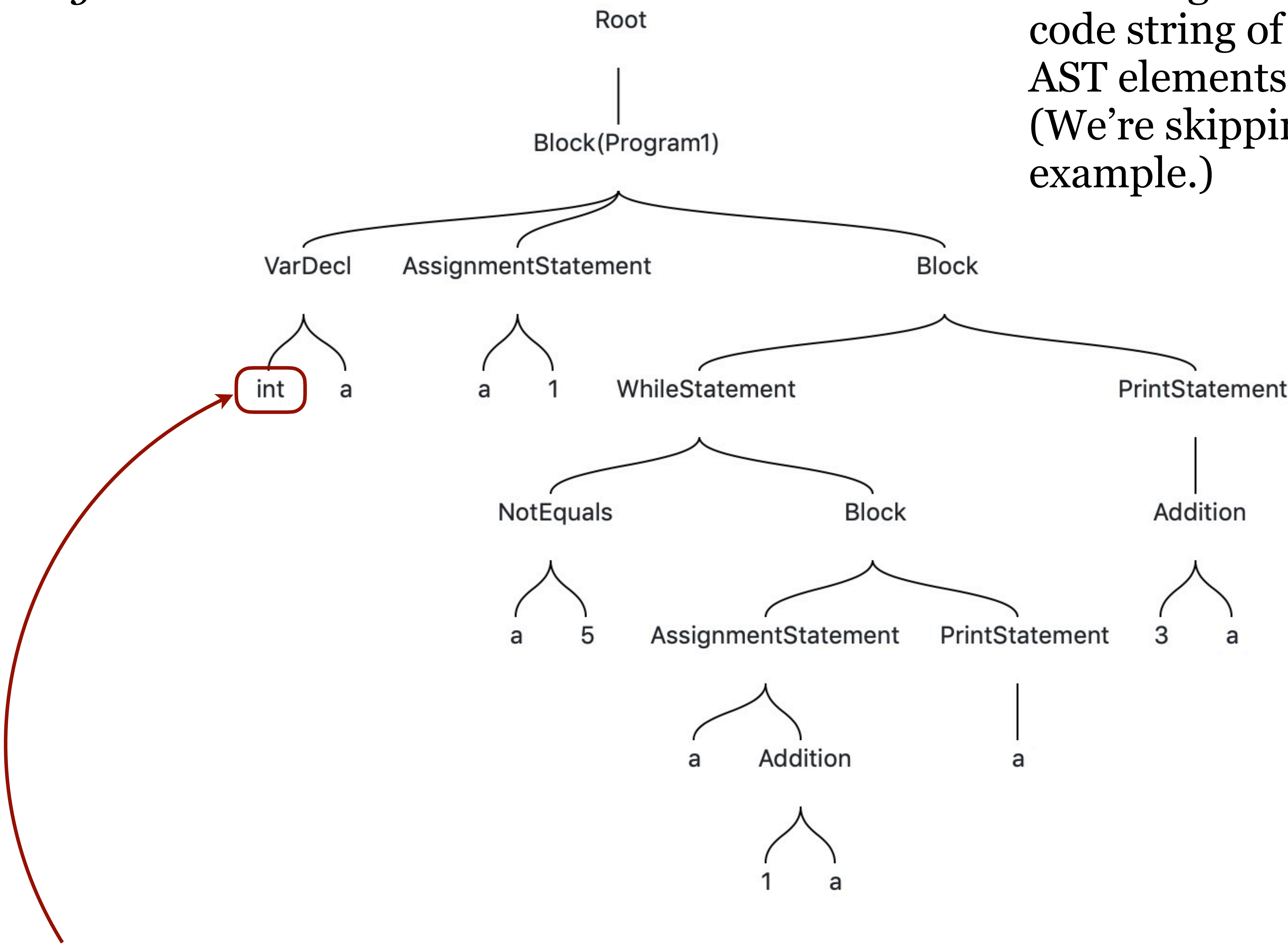



```
{intaa=1{while(a!=5){a=1+aprint(a)}print(3+a)}}$
```

Lexical Analysis

looking ahead . . .

Goal: to get from the source code string of characters to AST elements and structures. (We're skipping *Parse* in this example.)

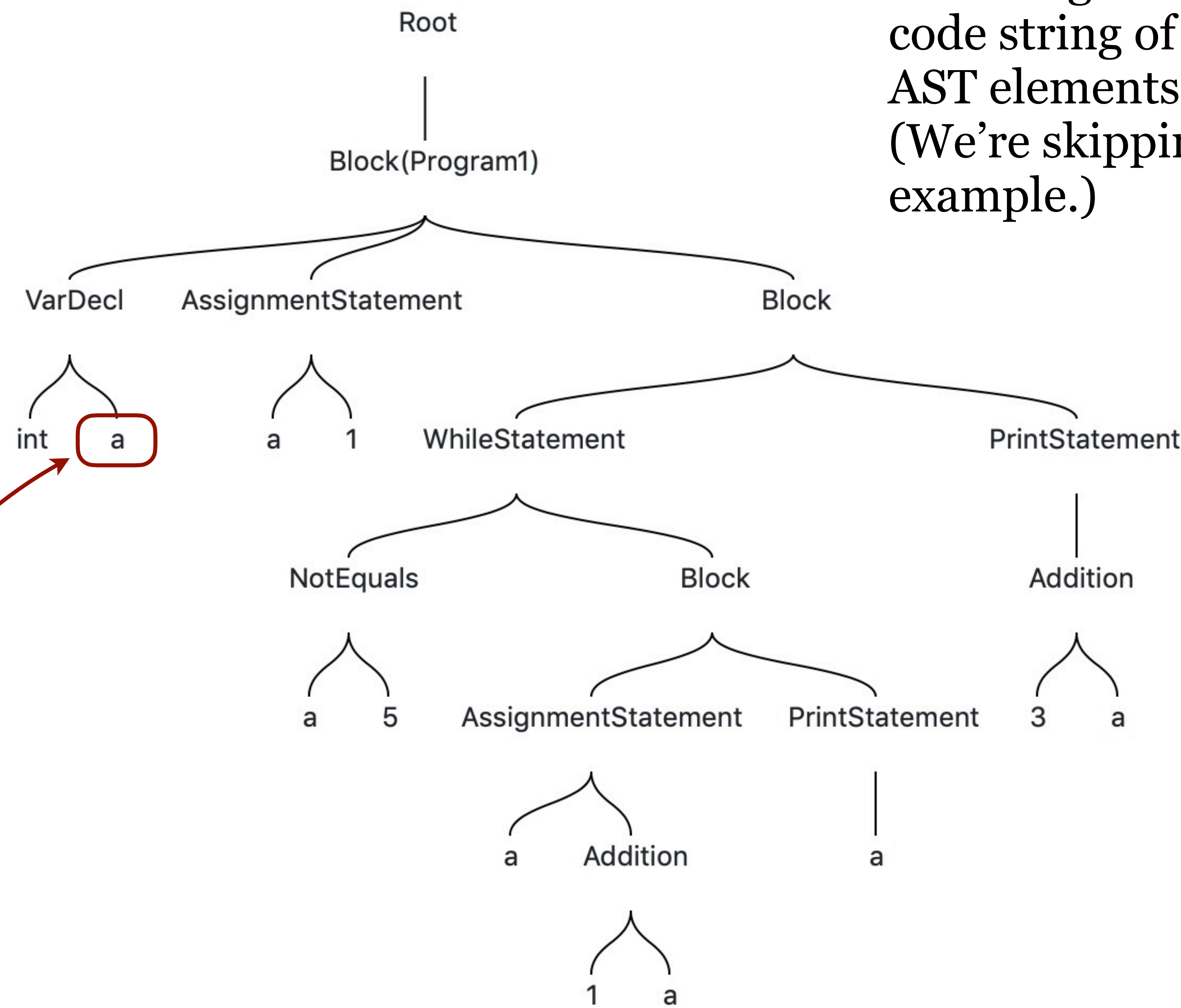


`{int aa=1{while(a!=5){a=1+aprint(a)}print(3+a)}}$`

Lexical Analysis

looking ahead . . .

Goal: to get from the source code string of characters to AST elements and structures. (We're skipping *Parse* in this example.)

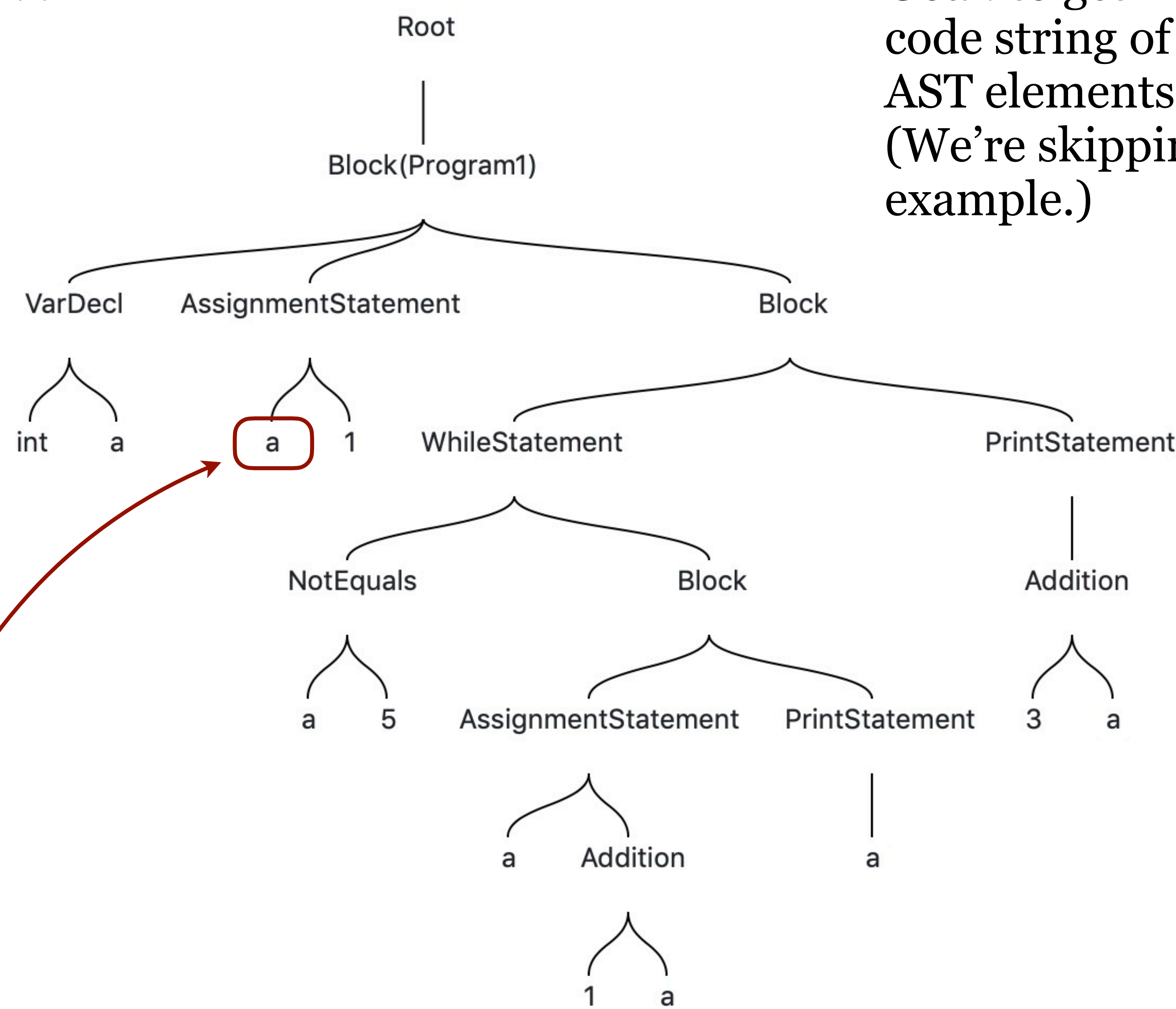



```
{inta a=1{while(a!=5){a=1+aprint(a)}print(3+a)}}$
```

Lexical Analysis

looking ahead . . .

Goal: to get from the source code string of characters to AST elements and structures. (We're skipping *Parse* in this example.)

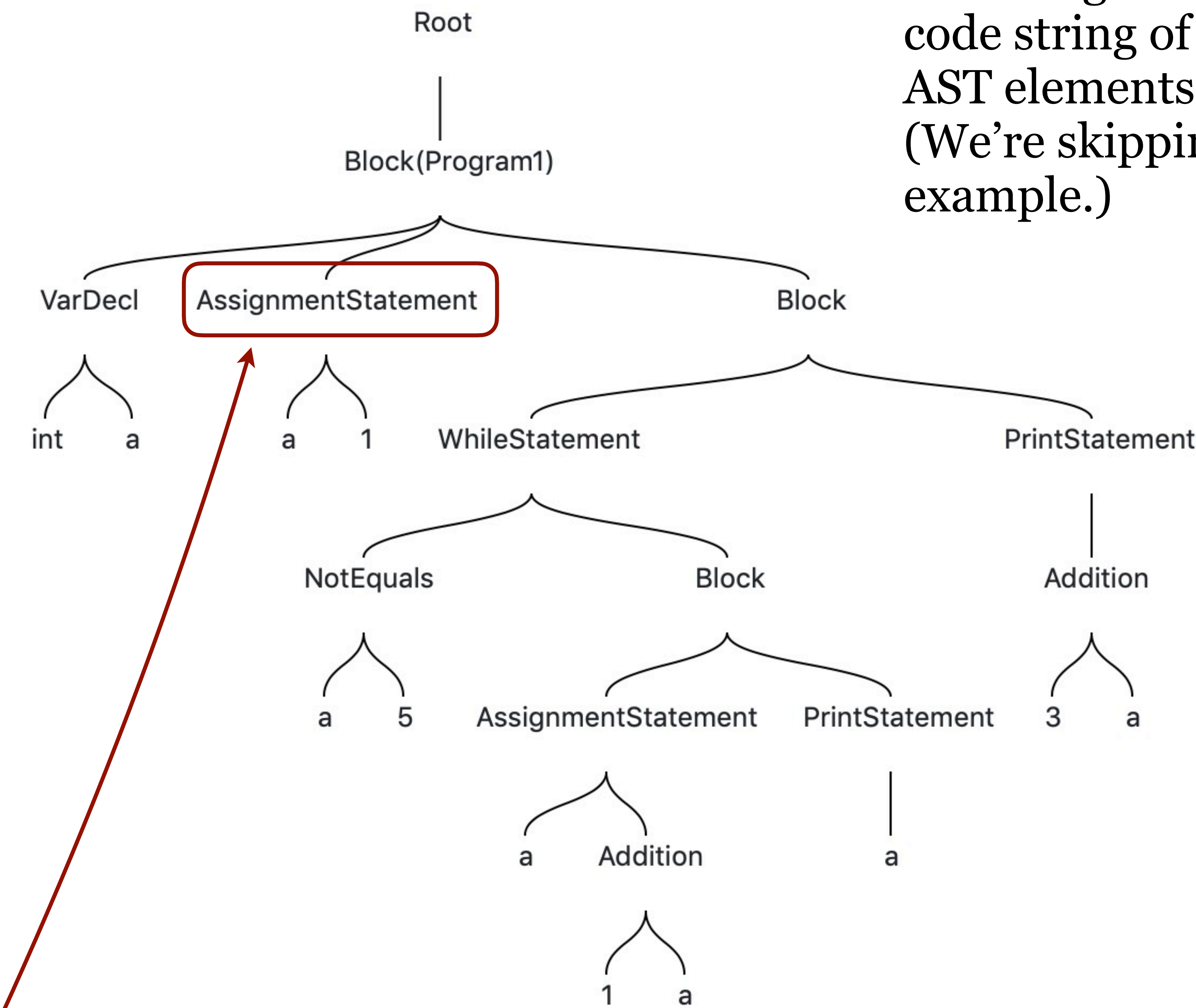



```
{inta a=1{while(a!=5){a=1+aprint(a)}print(3+a)}}$
```

Lexical Analysis

looking ahead . . .

Goal: to get from the source code string of characters to AST elements and structures. (We're skipping *Parse* in this example.)

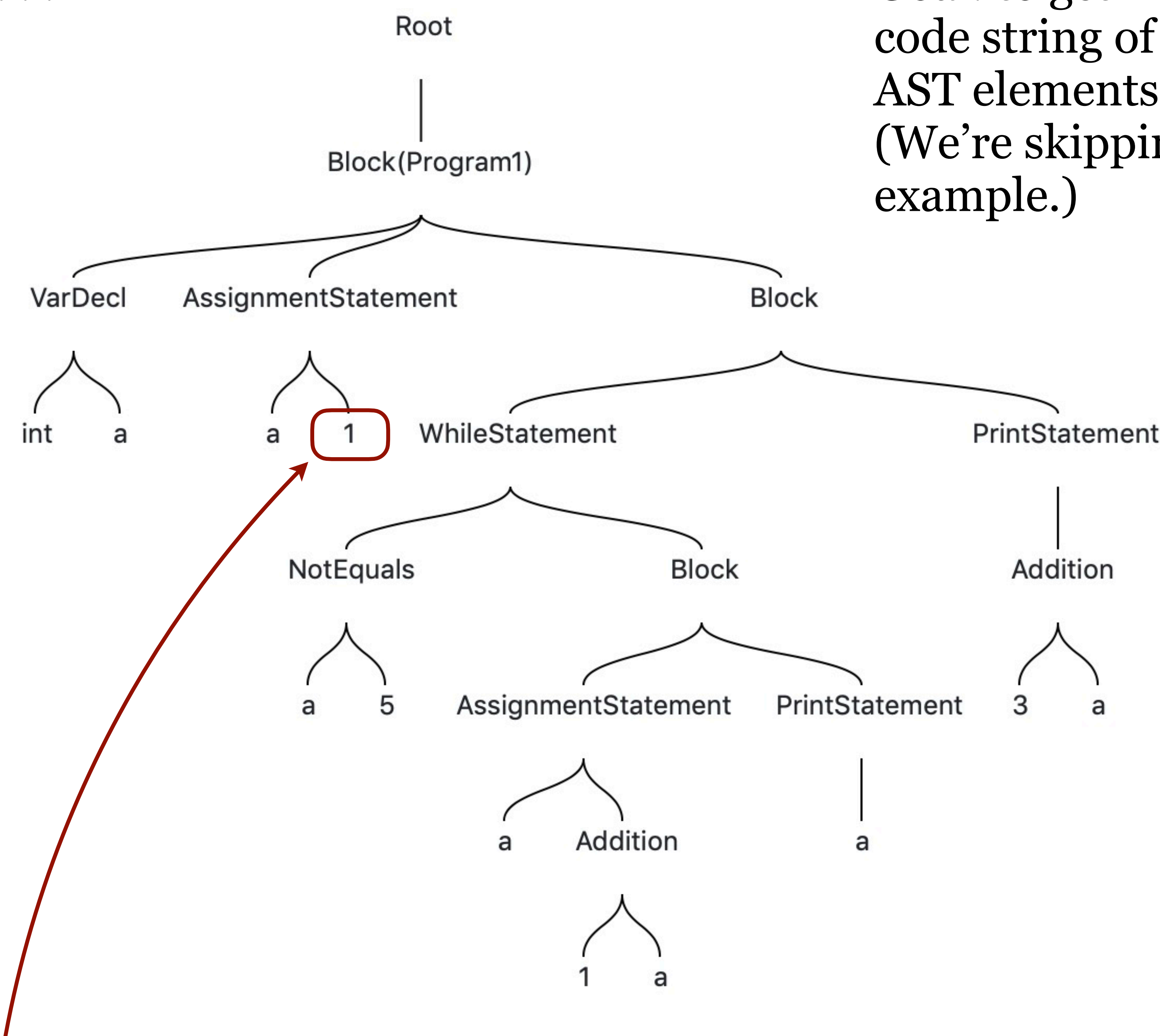



```
{intaa=1{while(a!=5){a=1+aprint(a)}print(3+a)}}$
```

Lexical Analysis

looking ahead . . .

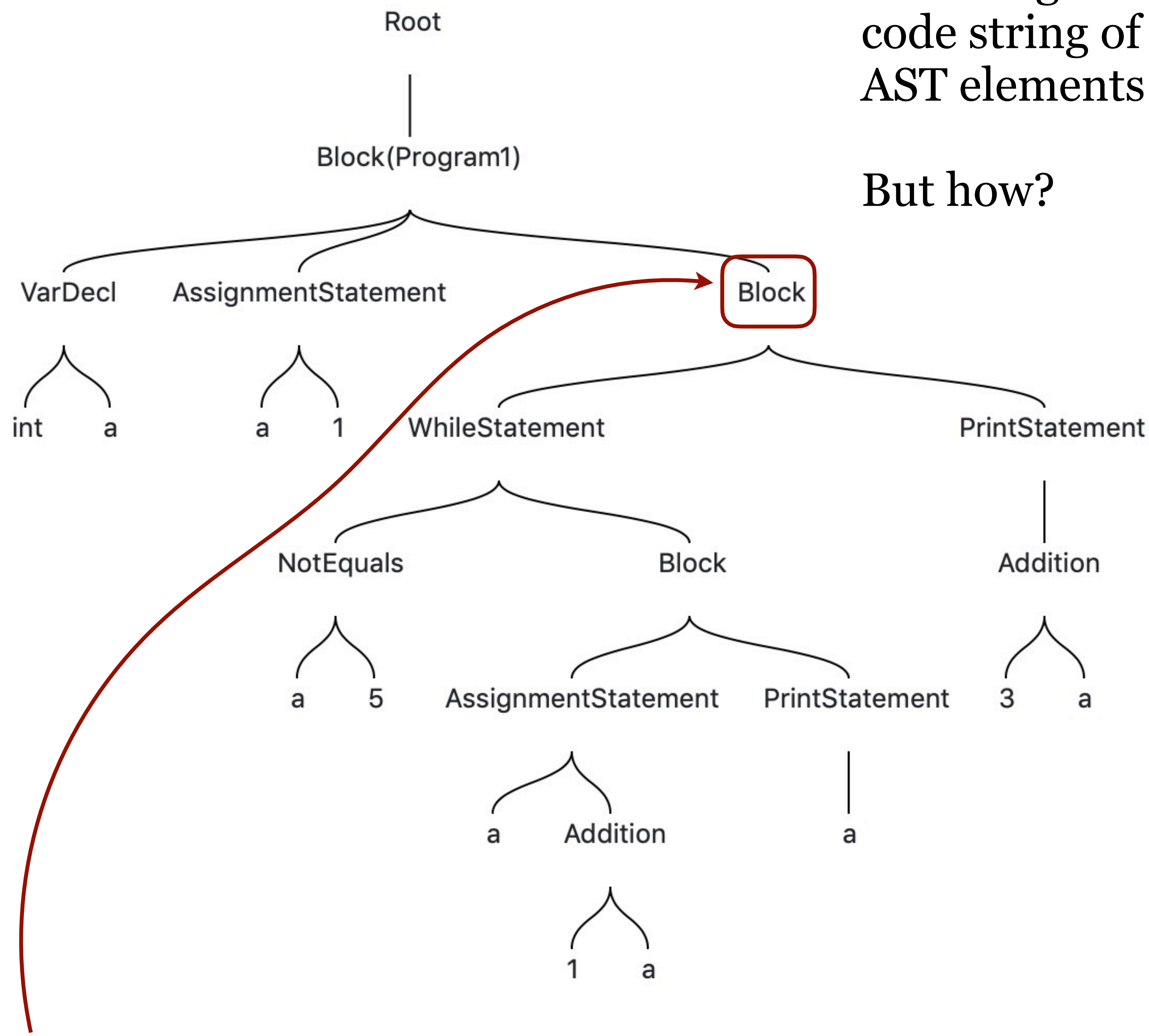
Goal: to get from the source code string of characters to AST elements and structures. (We're skipping *Parse* in this example.)




```
{intaa=1{while(a!=5){a=1+aprint(a)}print(3+a)}}$
```

Lexical Analysis

looking ahead . . .



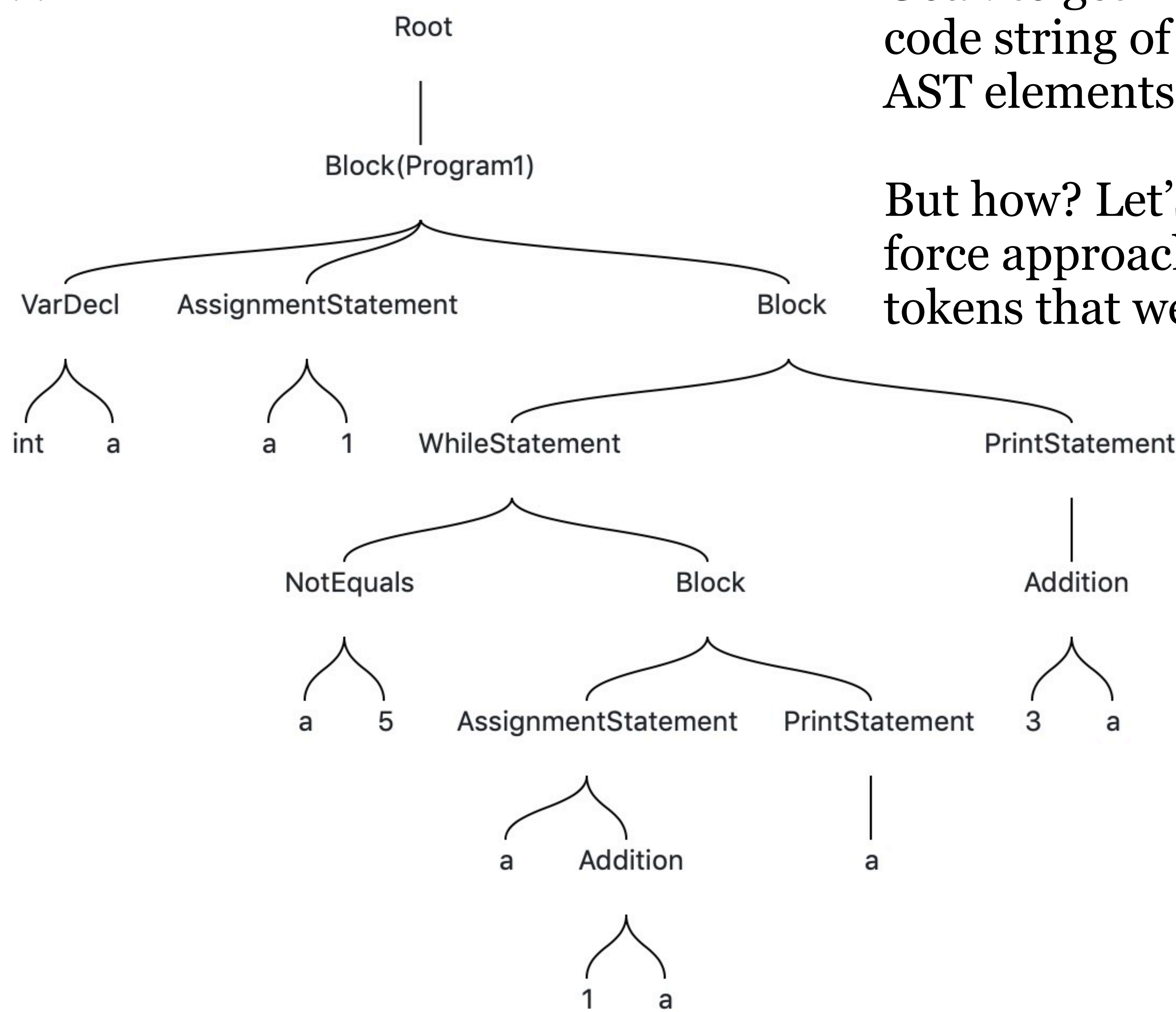
Goal: to get from the source code string of characters to AST elements and structures.

But how?

```
{intaa=1{while(a!=5){a=1+aprint(a)}print(3+a)}}$
```

Lexical Analysis

looking ahead . . .



Goal: to get from the source code string of characters to AST elements and structures.

But how? Let's try a brute force approach to making tokens that we'll send to *Parse*.

```
{intaa=1{while(a!=5){a=1+aprint(a)}print(3+a)}}$
```

Making Tokens Through Brute Force

Lex reads the source code character by character. We can process it the same way.

```
c = getNextChar();
if (c == 'c')  // class, close, case, catch, char, const
   c = getNextChar();
   if (c == 'l')  // class or close
      c = getNextChar();
      if (c == 'a')
         c = getNextChar();
         if (c == 's')
            c = getNextChar();
            if (c == 's')
               Token t = new Token('keyword_class');
            endif
         endif
      else if (c == "o") {
         // code to detect "close" keyword
      endif
   else if (c == 'a') // case or catch
      c = getNextChar();
      if (c == 's')
         c = getNextChar();
         if (c == 'e')
            Token t = new Token('keyword_case');
         endif
      else
       ⋮
      endif
   endif
else
 ⋮
endif
```

Making Tokens Through Brute Force

Lex reads the source code character by character. We can process it the same way.

```
c = getNextChar();
if (c == ‘c’)  // class, close, case, catch, char, const
   c = getNextChar();
   if (c == ‘l’)  // class or close
      c = getNextChar();
      if (c == ‘a’)
         c = getNextChar();
         if (c == ‘s’)
            c = getNextChar();
            if (c == ‘s’)
               Token t = new Token(‘keyword_class’);
            endif
         endif
      else if (c == “o”) {
         // code to detect “close” keyword
      endif
   else if (c == ‘a’) // case or catch
      c = getNextChar();
      if (c == ‘s’)
         c = getNextChar();
         if (c == ‘e’)
            Token t = new Token(‘keyword_case’);
         endif
      else
       ⋮
      endif
   endif
else
 ⋮
endif
```

This is a terrible idea.
It would be a nightmare to get to work at scale. You would be lucky to get it to work for even a tiny grammar.

There has **got** to be a better way.

Pattern Matching to Make Tokens

A lexical analyzer (sometimes called a *scanner*) must recognize all parts of the language's syntax: keywords, identifiers, symbols, digits, characters, and anything else in its lexicon.

We need to define these. Here are 3 examples:

identifier

integer

decimal

Pattern Matching to Make Tokens

A lexical analyzer (sometimes called a *scanner*) must recognize all parts of the language's syntax: keywords, identifiers, symbols, digits, characters, and anything else in its lexicon.

We need to define these. Here are 3 examples:

identifier
alphabetic followed by alphanumerics

integer

decimal

Pattern Matching to Make Tokens

A lexical analyzer (sometimes called a *scanner*) must recognize all parts of the language's syntax: keywords, identifiers, symbols, digits, characters, and anything else in its lexicon.

We need to define these. Here are 3 examples:

identifier
alphabetic followed by
alphanumeric**s**

integer
0 or (digit from 1 — 9 followed
by digit**s** from 0 — 9)

decimal

Pattern Matching to Make Tokens

A lexical analyzer (sometimes called a *scanner*) must recognize all parts of the language's syntax: keywords, identifiers, symbols, digits, characters, and anything else in its lexicon.

We need to define these. Here are 3 examples:

identifier
alphabetic followed by
alphanumeric**s**

integer
0 or (digit from 1 — 9 followed
by digit**s** from 0 — 9)

decimal
integer followed by '.'
followed by integer**s**

Pattern Matching to Make Tokens

A lexical analyzer (sometimes called a *scanner*) must recognize all parts of the language's syntax: keywords, identifiers, symbols, digits, characters, and anything else in its lexicon.

We need to define these. Here are 3 examples:

identifier
alphabetic followed by alphanumeric**s**

integer
0 or (digit from 1 — 9 followed by digit**s** from 0 — 9)

decimal
integer followed by '.' followed by integer**s**

These are nice, if slightly ambiguous.
But we need more than words to specify patterns if we are to write programs to do it.

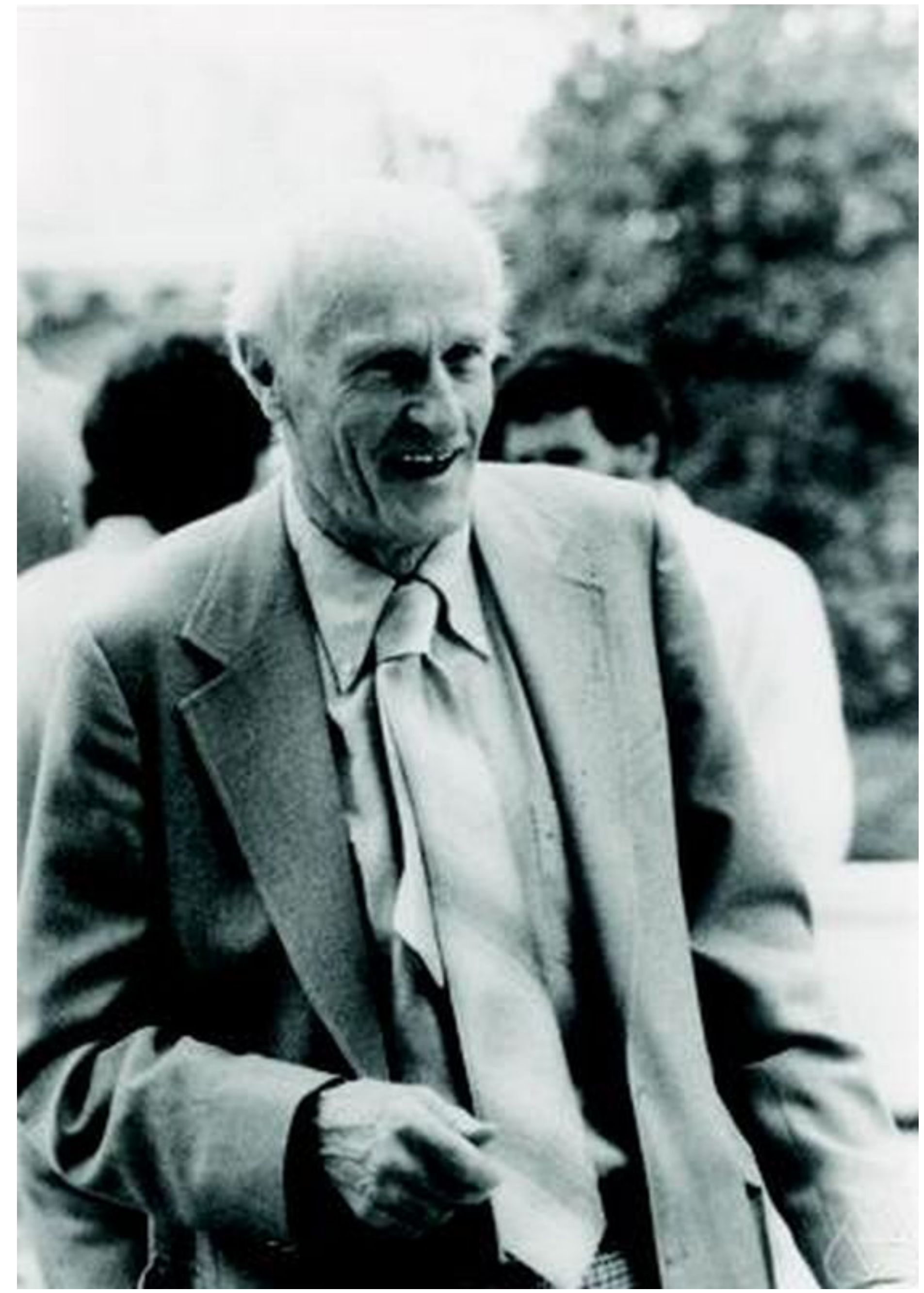
We need the power of . . .

Regular Expressions

A **RegEx** is a string that describes a set of other strings according to certain syntax rules.

- A common feature in many programming languages.
- Used to specify grammar formalities.
- Basis in/of Formal Languages and Automata Theory

Stephen Kleene ("KLAY-nee") described automata models with a mathematical notation called *regular sets*.



Stephen Kleene
1909 — 1994

Regular Expressions

Let $\mathbf{\Sigma} = \{a, b\}$

Uppercase Sigma, meaning the alphabet/lexicon of this language.

In this example, our entire ("regular") language consists only of *a*s and *b*s.

Definitions:

- A *formal language* (as opposed to a written, spoken, or programming language) is just a specifically-defined set of strings.
- An *alphabet* (or lexicon) is a finite set of symbols we'll call **Σ** (sigma).
  - e.g., $\mathbf{\Sigma} = \{0, 1\}$ is the binary alphabet/lexicon
- A *string* over an alphabet is a finite set of symbols drawn from **Σ**.
  - There is also the *empty string*, which we'll call **ε** (epsilon).
- A *language* is a set of strings that can be formed from the alphabet/lexicon.
- A *sentence* is a sequence of strings in the language.
- The ordering of strings within a sentence is defined by a set of rules called a *grammar*.

Now we can specify legal patterns expressible by our language.

Regular Expressions

Let $\mathbf{\Sigma} = \{a, b\}$

- $a|b$ means "a or b". $\{a, b\}$

Regular Expressions

Let $\mathbf{\Sigma} = \{a, b\}$

- $a|b$ means "a or b". $\{a, b\}$
- $(a|b)\ (a|b)$ means "a or b followed by a or b". $\{aa, ab, ba, bb\}$

Regular Expressions

Let $\mathbf{\Sigma} = \{a, b\}$

- $a|b$ means "a or b". $\{a, b\}$
- $(a|b)\ (a|b)$ means "a or b followed by a or b". $\{aa, ab, ba, bb\}$
- a^* means "zero or more as". $\{\varepsilon, a, aa, aaa, aaaa, \ldots\}$

Lowercase epsilon meaning the empty string. (Some books use λ instead. We'll stick to ε.)

Regular Expressions

Let $\mathbf{\Sigma} = \{a, b\}$

- $a|b$ means "a or b". $\{a, b\}$
- $(a|b)\,(a|b)$ means "a or b followed by a or b". $\{aa, ab, ba, bb\}$
- a^* means "zero or more as". $\{\varepsilon, a, aa, aaa, aaaa, \ldots\}$
- a^+ means "one or more as". $\{a, aa, aaa, aaaa, \ldots\}$

Regular Expressions

Let $\mathbf{\Sigma} = \{a, b\}$

- $a|b$ means "a or b". $\{a, b\}$
- $(a|b)\ (a|b)$ means "a or b followed by a or b". $\{aa, ab, ba, bb\}$
- a^* means "zero or more as". $\{\varepsilon, a, aa, aaa, aaaa, \ldots\}$
- a^+ means "one or more as". $\{a, aa, aaa, aaaa, \ldots\}$

Other Examples

- $(a|b)^*$ means "all strings of as and bs including ε".
- $(a^*|b^*)^*$ also means "all strings of as and bs including ε".
- $a|(a^*b)$ denotes $\{a, b, ab, aab, aaab, aaaab, \ldots\}$

Regular Expressions

Let **Σ** = {0,1,2,3,4,5,6,7,8,9,+,-}

- Even numbers
  - (+|-|ε) (0|1|2|3|4|5|6|7|8|9)* (0|2|4|6|8)

Regular Expressions

Let **Σ** = {0,1,2,3,4,5,6,7,8,9,+,-}

- Even numbers
  - (+|-|**ε**) (0|1|2|3|4|5|6|7|8|9)* (0|2|4|6|8)

- We can make our own definitions
  - Sign = + | —
  - OptionalSign = Sign | **ε**
  - Digit = 0|1|2|3|4|5|6|7|8|9
  - EvenDigit = 0|2|4|6|8

  - EvenNumber = OptionalSign Digit* EvenDigit

    "OptionalSign followed by zero or more Digit**s** followed by an EvenDigit"

Regular Expressions

Let **Σ** = {0,1,2,3,4,5,6,7,8,9,+,-}

- Even numbers
  - (+|-|**ε**) (0|1|2|3|4|5|6|7|8|9)* (0|2|4|6|8)

- We can make our own definitions
  - Sign = + | —
  - OptionalSign = Sign | **ε**
  - Digit = **[0123456789]**
  - EvenDigit = **[02468]**

  Multi-way Disjunction (“or”)

  - EvenNumber = OptionalSign Digit* EvenDigit

Regular Expressions

Let **Σ** = {0,1,2,3,4,5,6,7,8,9,+,-}

- Even numbers
  - (+|-|**ε**) (0|1|2|3|4|5|6|7|8|9)* (0|2|4|6|8)

- We can make our own definitions
  - Sign = + | —
  - OptionalSign = Sign | **ε**
  - Digit = **[0-9]** ← Multi-way Disjunction over a range
  - EvenDigit = [02468]

  - EvenNumber = OptionalSign Digit* EvenDigit

Regular Expressions

Let **Σ** = {0,1,2,3,4,5,6,7,8,9,+,-}

- Even numbers
  - (+|-|ε) (0|1|2|3|4|5|6|7|8|9)* (0|2|4|6|8)

- We can make our own definitions
  - Sign = + | —
  - OptionalSign = Sign**?** ← **?** means "zero or one".
  - Digit = [0-9]
  - EvenDigit = [02468]

  - EvenNumber = OptionalSign Digit* EvenDigit

Pattern Matching to Make Tokens

Let's revisit our wordy definitions and define them more precisely.

identifier
alphabetic followed by
alphanumeric**s**

integer
0 or (digit from 1 — 9 followed
by digit**s** from 0 — 9)

decimal
integer followed by '.'
followed by integer**s**

Pattern Matching to Make Tokens

Let's revisit our wordy definitions and define them more precisely.

identifier
alphabetic followed by
0 or more alphanumerics

[a-z][a-z0-9]*

integer
0 or (digit from 1 — 9 followed by digit**s** from 0 — 9)

decimal
integer followed by '.' followed by integer**s**

Pattern Matching to Make Tokens

Let's revisit our wordy definitions and define them more precisely.

identifier
alphabetic followed by
0 or more alphanumerics

[a-z][a-z0-9]*

integer
0 or (digit from 1 — 9 followed
by 0 or more digits from 0 — 9)

0|([1-9][0-9]*)

decimal
integer followed by '.'
followed by integer**s**

Pattern Matching to Make Tokens

Let's revisit our wordy definitions and define them more precisely.

identifier
alphabetic followed by
0 or more alphanumerics

[a-z][a-z0-9]*

integer
0 or (digit from 1 — 9 followed by 0 or more digits from 0 — 9)

0|([1-9][0-9]*)

Why not just $[0\text{-}9]^+$?

decimal
integer followed by '.'
followed by integer**s**

Pattern Matching to Make Tokens

Let's revisit our wordy definitions and define them more precisely.

identifier
alphabetic followed by
0 or more alphanumerics

[a-z][a-z0-9]*

integer
0 or (digit from 1 — 9 followed
by 0 or more digits from 0 — 9)

0|([1-9][0-9]*)

decimal
integer followed by '.'
followed by 0 or more integers

(0|([1-9][0-9]*)) '.' (0|([1-9][0-9]*))*

Pattern Matching to Make Tokens

Let's revisit our wordy definitions and define them more precisely.

identifier
alphabetic followed by
0 or more alphanumerics

[a-z][a-z0-9]*

integer
0 or (digit from 1 — 9 followed by 0 or more digits from 0 — 9)

0|([1-9][0-9]*)

decimal
integer followed by '.'
followed by 0 or more integers
or
followed by 1 or more integers

(0|([1-9][0-9]*)) '.' (0|([1-9][0-9]*))*

(0|([1-9][0-9]*)) '.' (0|([1-9][0-9]*))+

Which one?

Pattern Matching to Make Tokens

Let's revisit our wordy definitions and define them more precisely.

Design-time choices like these have a huge impact on compiler implementation and the way the programming language is used by its programmers.

identifier
alphabetic followed by
0 or more alphanumerics

[a-z][a-z0-9]*

integer
0 or (digit from 1 — 9 followed by 0 or more digits from 0 — 9)

0|([1-9][0-9]*)

Why not just [0-9]+ ?

decimal
integer followed by '.'
followed by 0 or more integers
or
followed by 1 or more integers

(0|([1-9][0-9]*)) '.' (0|([1-9][0-9]*))*

(0|([1-9][0-9]*)) '.' (0|([1-9][0-9]*))+

Which one?

Lexical Analysis

Example:

```
int f int if=1
```

Given patterns that define keywords, identifiers, symbols, digits, characters, how does Lex categorize this sentence?

Remember, spaces can be helpful but are not required. They may even be misleading (if a mean professor is trying to trick you).

Lexical Analysis

Example:

```
int f int if=1
```

Given patterns that define keywords, identifiers, symbols, digits, characters, how does Lex categorize this sentence?

Remember, spaces can be helpful but are not required. They may even be misleading (if a mean professor is trying to trick you).

`int`	`f`	`int`	`if`	`=`	`1`
keyword	id	keyword	keyword	assign	int

Lexical Analysis

Example:

```
int f int if=1
```

Given patterns that define keywords, identifiers, symbols, digits, characters, how does Lex categorize this sentence?

Remember, spaces can be helpful but are not required. They may even be misleading (if a mean professor is trying to trick you).

```
int f int if = 1
```

or

```
int f int i f = 1
```

keyword id keyword id id assign int

Lexical Analysis

Example:

```
int f int if=1
```

Given patterns that define keywords, identifiers, symbols, digits, characters, how does Lex categorize this sentence?

Remember, spaces can be helpful but are not required. They may even be misleading (if a mean professor is trying to trick you).

```
int f int if=1
```

or

```
int f int if=1
```

Q: How do we know?
A: Two steps:

1. Longest Match
2. Rule Order

Lexical Analysis

Example: Longest Match (a "greedy" approach)

```
int f int if=1
```

When Lex is scanning character by character, take as many characters as you can when looking for patterns to match.

For example, scanning the above from the beginning;

`i`	Matched an id, but keep looking.
`in`	No new match, so we still think it's an id.
`int`	Matches a keyword, a longer match than id, so update to keyword.
space	We can use this as a separator and stop looking.

Now we emit the longest match (keyword) and continue lexing from the next char.

Lexical Analysis

Example: Longest Match (a "greedy" approach)

`int f int if=1`

When Lex is scanning character by character, take as many characters as you can when looking for patterns to match.

For example, scanning the above from the beginning;

`i`	Matched an id, but keep looking.
`in`	No new match, so we still think it's an id.
`int`	Matches a keyword, a longer match than id, so update to keyword.
space	We can use this as a separator and stop looking.

Now we emit the longest match (keyword) and continue lexing from the next char.

This means that our interpretation of the above is

`int`	**`f`**	**`int`**	**`if`**	**`=`**	**`1`**
keyword	id	keyword	keyword	assign	int

Lexical Analysis

Example: Rule Order

```
int f int if=1
```

Once we have the longest match to determine the token, the order in which we specify the lexical rules defines their precedence. For our language in this class, the order is:

1. keyword
2. id
3. symbol
4. digit
5. char

This still means that our interpretation of the above is

```
int f int if=1
```

keyword id keyword keyword assign int

Lexical Analysis

Example of bad language design:

```
int f int if=1
```

We can make a good argument for statement separators based on this example. Requiring a semicolon (for example) to end every statement removes this ambiguity.

```
int f; int i;f=1
```

Lexical Analysis

Detailed example with resources on our web site:

- Look at our language grammar, then
- the example Lex Without Spaces.

```
/*LongTestCase-EverythingExceptBooleanDeclaration*/{/*IntDeclaration*/
intaintba=0b=0/*WhileLoop*/while(a!=3){print(a)while(b!=3){print(b)b=1+bif(b==2)
{/*PrintStatement*/print("there is no spoon"/*Thiswilldonothing*/)}}b=0a=1+a}}$
```

```
LEXER --> | T_OPENING_BRACE [ { ]  on line 1...
LEXER --> | T_VARIABLE_TYPE [ int ]  on line 1...
LEXER --> | T_ID [ a ]  on line 1...
LEXER --> | T_VARIABLE_TYPE [ int ]  on line 1...
LEXER --> | T_ID [ b ]  on line 1...
LEXER --> | T_ID [ a ]  on line 1...
LEXER --> | T_ASSIGNMENT_OP [ = ]  on line 1...
LEXER --> | T_DIGIT [ 0 ]  on line 1...
LEXER --> | T_ID [ b ]  on line 1...
LEXER --> | T_ASSIGNMENT_OP [ = ]  on line 1...
LEXER --> | T_DIGIT [ 0 ]  on line 1...
LEXER --> | T_WHILE [ while ]  on line 1...
LEXER --> | T_OPENING_PARENTHESIS [ ( ]  on line 1...
LEXER --> | T_ID [ a ]  on line 1...
LEXER --> | T_INEQUALITY_OP [ != ]  on line 1...
LEXER --> | T_DIGIT [ 3 ]  on line 1...
LEXER --> | T_CLOSING_PARENTHESIS [ ) ]  on line 1...
LEXER --> | T_OPENING_BRACE [ { ]  on line 1...
LEXER --> | T_PRINT [ print ]  on line 1...
LEXER --> | T_OPENING_PARENTHESIS [ ( ]  on line 1...
LEXER --> | T_ID [ a ]  on line 1...
LEXER --> | T_CLOSING_PARENTHESIS [ ) ]  on line 1...
LEXER --> | T_WHILE [ while ]  on line 1...
LEXER --> | T_OPENING_PARENTHESIS [ ( ]  on line 1...
LEXER --> | T_ID [ b ]  on line 1...
LEXER --> | T_INEQUALITY_OP [ != ]  on line 1...
LEXER --> | T_DIGIT [ 3 ]  on line 1...
LEXER --> | T_CLOSING_PARENTHESIS [ ) ]  on line 1...
LEXER --> | T_OPENING_BRACE [ { ]  on line 1...
LEXER --> | T_PRINT [ print ]  on line 1...
LEXER --> | T_OPENING_PARENTHESIS [ ( ]  on line 1...
LEXER --> | T_ID [ b ]  on line 1...
LEXER --> | T_CLOSING_PARENTHESIS [ ) ]  on line 1...
LEXER --> | T_ID [ b ]  on line 1...
LEXER --> | T_ASSIGNMENT_OP [ = ]  on line 1...
LEXER --> | T_DIGIT [ 1 ]  on line 1...
LEXER --> | T_ADDITION_OP [ + ]  on line 1...
LEXER --> | T_ID [ b ]  on line 1...
LEXER --> | T_IF [ if ]  on line 1...
LEXER --> | T_OPENING_PARENTHESIS [ ( ]  on line 1...
LEXER --> | T_ID [ b ]  on line 1...
LEXER --> | T_EQUALITY_OP [ == ]  on line 1...
LEXER --> | T_DIGIT [ 2 ]  on line 1...
LEXER --> | T_CLOSING_PARENTHESIS [ ) ]  on line 1...
LEXER --> | T_OPENING_BRACE [ { ]  on line 1...
LEXER --> | T_PRINT [ print ]  on line 1...
LEXER --> | T_OPENING_PARENTHESIS [ ( ]  on line 1...
LEXER --> | T_QUOTE [ " ]  on line 1...
LEXER --> | T_CHAR [ t ]  on line 1...
LEXER --> | T_CHAR [ h ]  on line 1...
LEXER --> | T_CHAR [ e ]  on line 1...
LEXER --> | T_CHAR [ r ]  on line 1...
LEXER --> | T_CHAR [ e ]  on line 1...
LEXER --> | T_CHAR [   ]  on line 1...
LEXER --> | T_CHAR [ i ]  on line 1...
LEXER --> | T_CHAR [ s ]  on line 1...
LEXER --> | T_CHAR [   ]  on line 1...
LEXER --> | T_CHAR [ n ]  on line 1...
LEXER --> | T_CHAR [ o ]  on line 1...
LEXER --> | T_CHAR [   ]  on line 1...
LEXER --> | T_CHAR [ s ]  on line 1...
LEXER --> | T_CHAR [ p ]  on line 1...
LEXER --> | T_CHAR [ o ]  on line 1...
LEXER --> | T_CHAR [ o ]  on line 1...
LEXER --> | T_CHAR [ n ]  on line 1...
LEXER --> | T_QUOTE [ " ]  on line 1...
LEXER --> | T_CLOSING_PARENTHESIS [ ) ]  on line 1...
LEXER --> | T_CLOSING_BRACE [ } ]  on line 1...
LEXER --> | T_CLOSING_BRACE [ } ]  on line 1...
LEXER --> | T_ID [ b ]  on line 1...
LEXER --> | T_ASSIGNMENT_OP [ = ]  on line 1...
LEXER --> | T_DIGIT [ 0 ]  on line 1...
LEXER --> | T_ID [ a ]  on line 1...
LEXER --> | T_ASSIGNMENT_OP [ = ]  on line 1...
LEXER --> | T_DIGIT [ 1 ]  on line 1...
LEXER --> | T_ADDITION_OP [ + ]  on line 1...
LEXER --> | T_ID [ a ]  on line 1...
LEXER --> | T_CLOSING_BRACE [ } ]  on line 1...
LEXER --> | T_CLOSING_BRACE [ } ]  on line 1...
LEXER --> | T_EOPS [ $ ]  on line 1...
```

Our Language Grammar

```
Program              ::== Block $
Block                ::== { StatementList  }
StatementList        ::== Statement StatementList
                     ::== ε
Statement            ::== PrintStatement
                     ::== AssignmentStatement
                     ::== VarDecl
                     ::== WhileStatement
                     ::== IfStatement
                     ::== Block
PrintStatement       ::== print ( Expr )
AssignmentStatement  ::== Id = Expr
VarDecl              ::== type Id
WhileStatement       ::== while BooleanExpr Block
IfStatement          ::== if BooleanExpr Block

Expr                 ::== IntExpr
                     ::== StringExpr
                     ::== BooleanExpr
                     ::== Id
IntExpr              ::== digit intop Expr
                     ::== digit
StringExpr           ::== " CharList "
BooleanExpr          ::== ( Expr boolop Expr )
                     ::== boolval
Id                   ::== char
CharList             ::== char CharList
                     ::== space CharList
                     ::== ε

type                 ::== int | string | boolean
char                 ::== a | b | c ... z
space                ::== the space character
digit                ::== 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

boolop               ::== == | !=
boolval              ::== false | true
intop                ::== +

Comments are bounded by /* and */ and ignored by the lexer.
```

Curly braces denote scope.

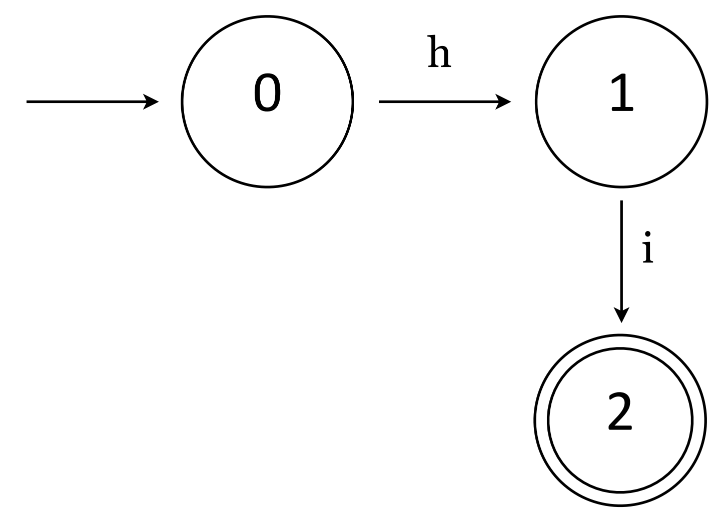
= is assignment.

== is test for equality.

Finite Automata

More that just stylized flowcharts for implementing transition diagrams, Finite Automata (FA) are actually graphs, and as such consist of vertices (circles) and directed edges (arrows).

"The word automaton, closely related to the word *automation*, denotes automatic processes carrying out the production of specific processes. Simply stated, automata theory deals with the **logic of computation** with respect to simple machines, referred to as *automata*. Through automata, computer scientists are able to understand how machines compute functions and solve problems.



From *https://cs.stanford.edu/people/eroberts/courses/soco/projects/2004-05/automata-theory/basics.html*
Emphasis added.

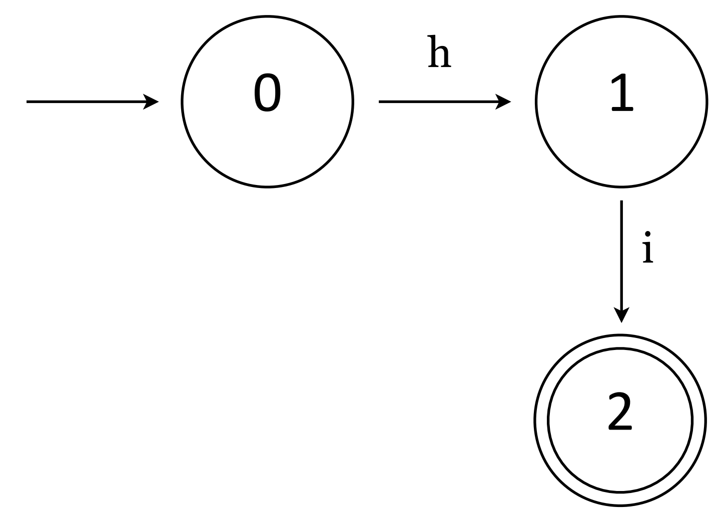
Finite Automata

States (vertices, circles)

- Each represents a possible moment/condition that could occur while scanning the input looking for a pattern to match so we can emit a token.
- The **start state** is denoted by an arrow from nowhere pointing to it.
- Accepting states are denoted by a double-circle. When we reach one we have matched a pattern, found a lexeme, and can emit a token.

Transitions (directed edges, arrows)

- Each edge is labeled with a value.
- If we are in a state and get as input a value matching an edge label then we consume that input and follow that edge, transitioning to the next state.
- If all edges leading from a given state are non-empty and disjoint then we have a Deterministic Finite Automata (DFA).



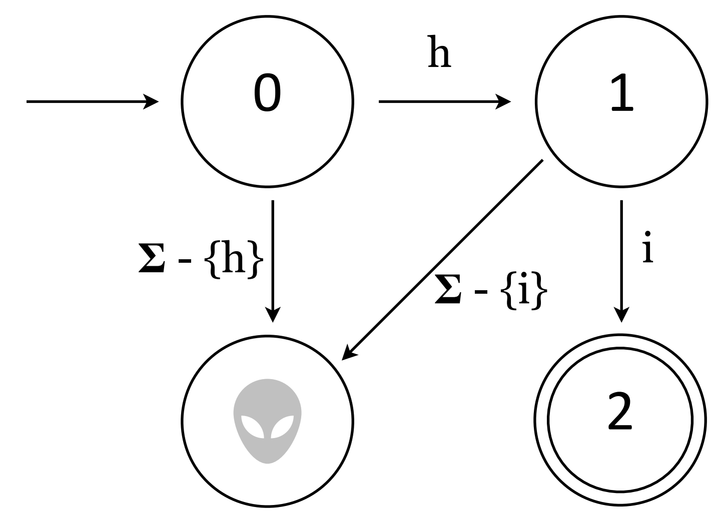
Finite Automata

States (vertices, circles)

- Each represents a possible moment/condition that could occur while scanning the input looking for a pattern to match so we can emit a token.
- The **start state** is denoted by an arrow from nowhere pointing to it.
- Accepting states are denoted by a double-circle. When we reach one we have matched a pattern, found a lexeme, and can emit a token.

Transitions (directed edges, arrows)

- Each edge is labeled with a value.
- If we are in a state and get as input a value matching an edge label then we consume that input and follow that edge, transitioning to the next state.
- If all edges leading from a given state are non-empty and disjoint then we have a Deterministic Finite Automata (DFA).



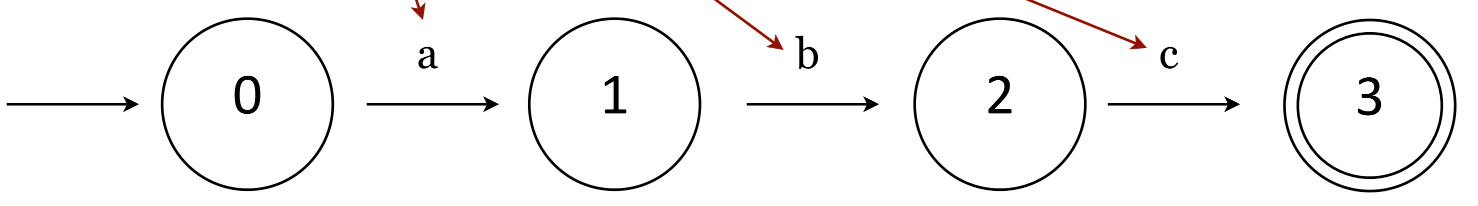
Formally, DFAs are required to account for **all** legal transitions, so we should include an error state.

Deterministic Finite Automata

Let $\mathbf{\Sigma} = \{a, b, c\}$

RegEx: $(a\ b\ c^{+})^{+}$

DFA:

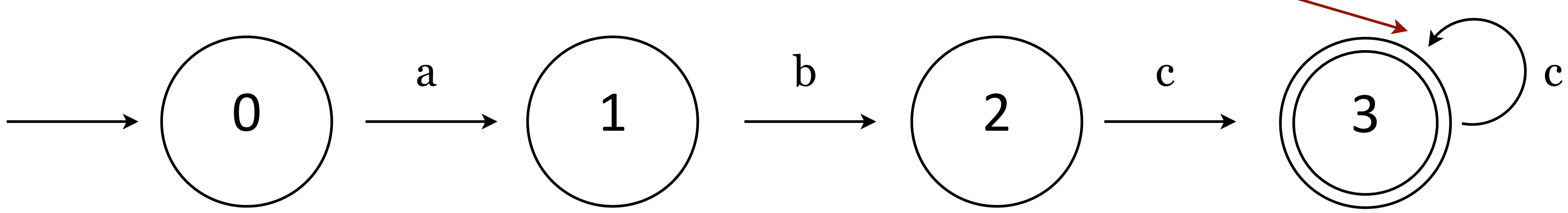


Deterministic Finite Automata

Let $\mathbf{\Sigma} = \{a, b, c\}$

RegEx: $(a\ b\ c^{+})^{+}$

DFA:

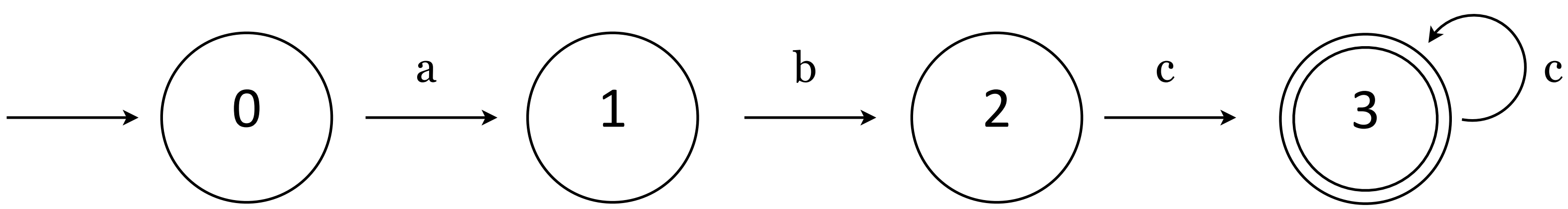


Deterministic Finite Automata

Let $\mathbf{\Sigma} = \{a, b, c\}$

RegEx: $(a\ b\ c^{+})^{\boxed{+}}$

DFA:

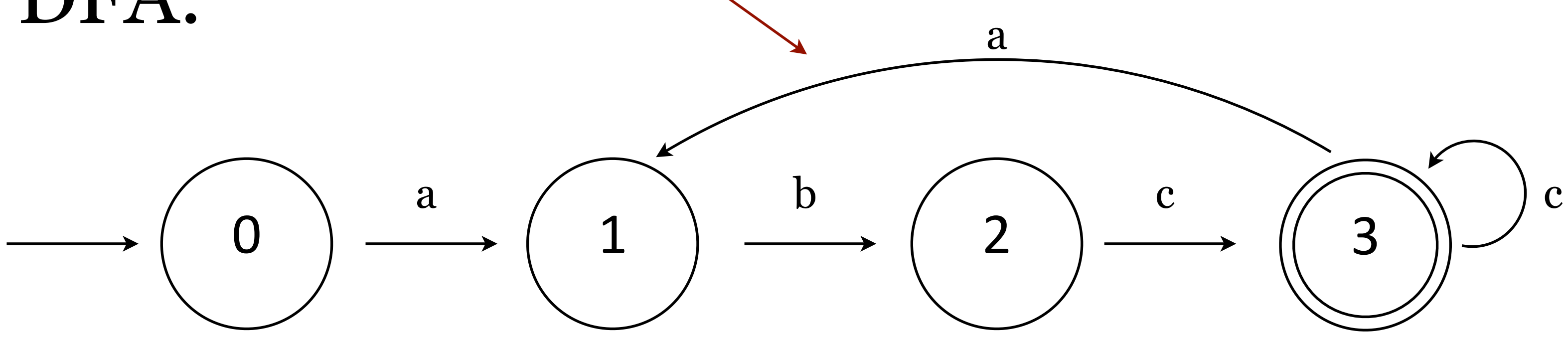


Deterministic Finite Automata

Let $\mathbf{\Sigma} = \{a, b, c\}$

RegEx: $(a\ b\ c^{+})^{+}$

DFA:

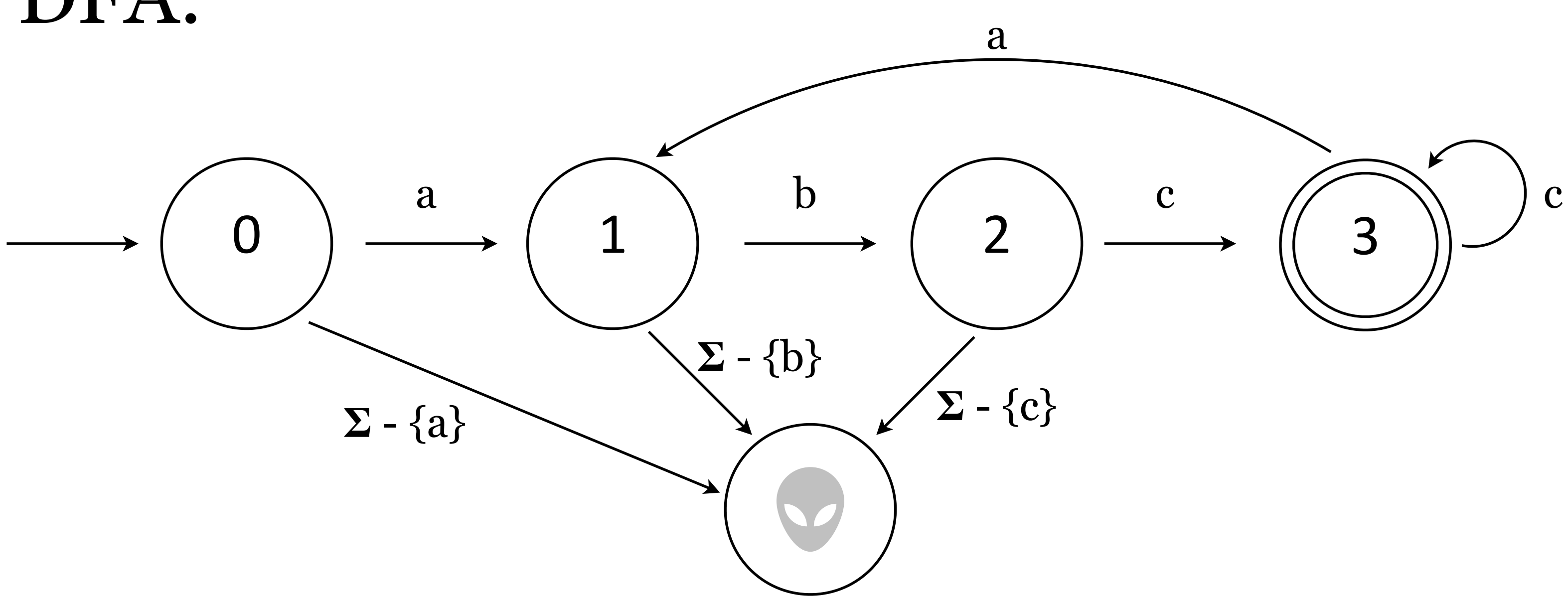


Deterministic Finite Automata

Let $\mathbf{\Sigma} = \{a, b, c\}$

RegEx: $(a\ b\ c^{+})^{+}$

DFA:



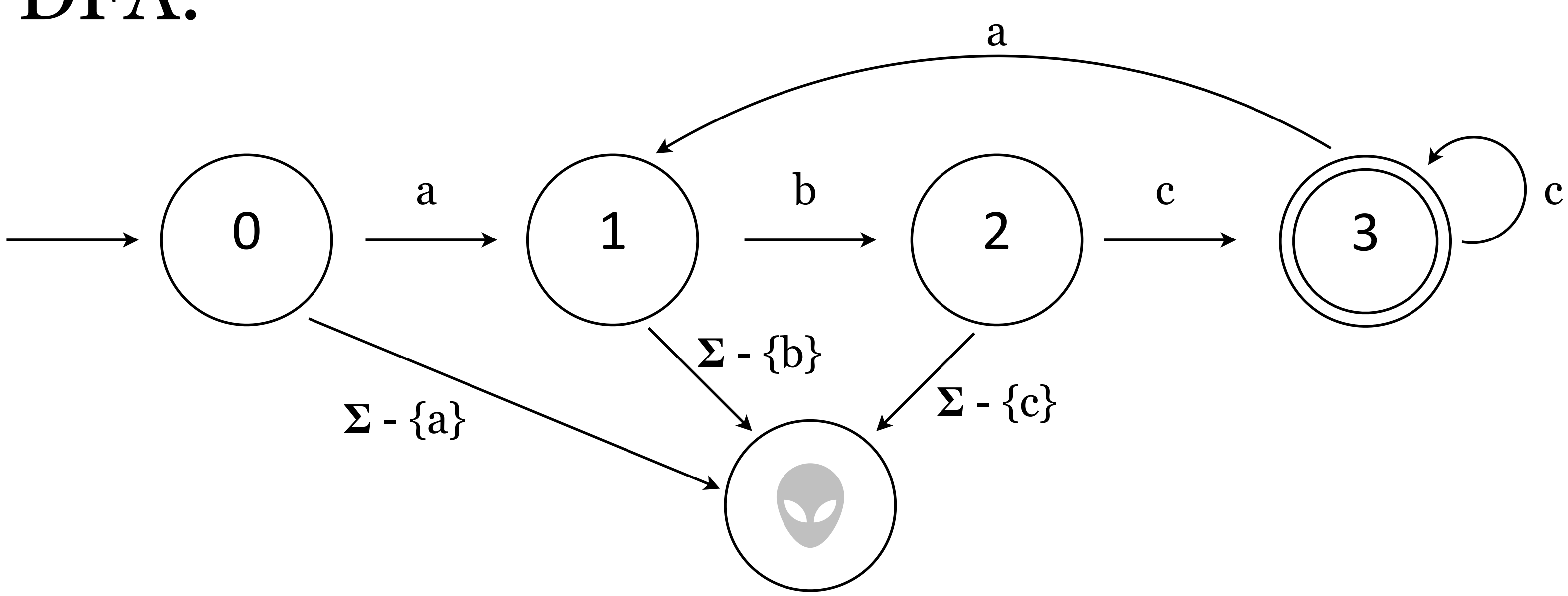
We're not going to include error states after this. From now on assume that if there is no valid transition out of a non-accepting state then it's an error.

Deterministic Finite Automata

Let $\mathbf{\Sigma} = \{a, b, c\}$

RegEx: $(a\ b\ c^{+})^{+}$

DFA:



Transition Table

	a	**b**	**c**
0			
1			
2			
3			

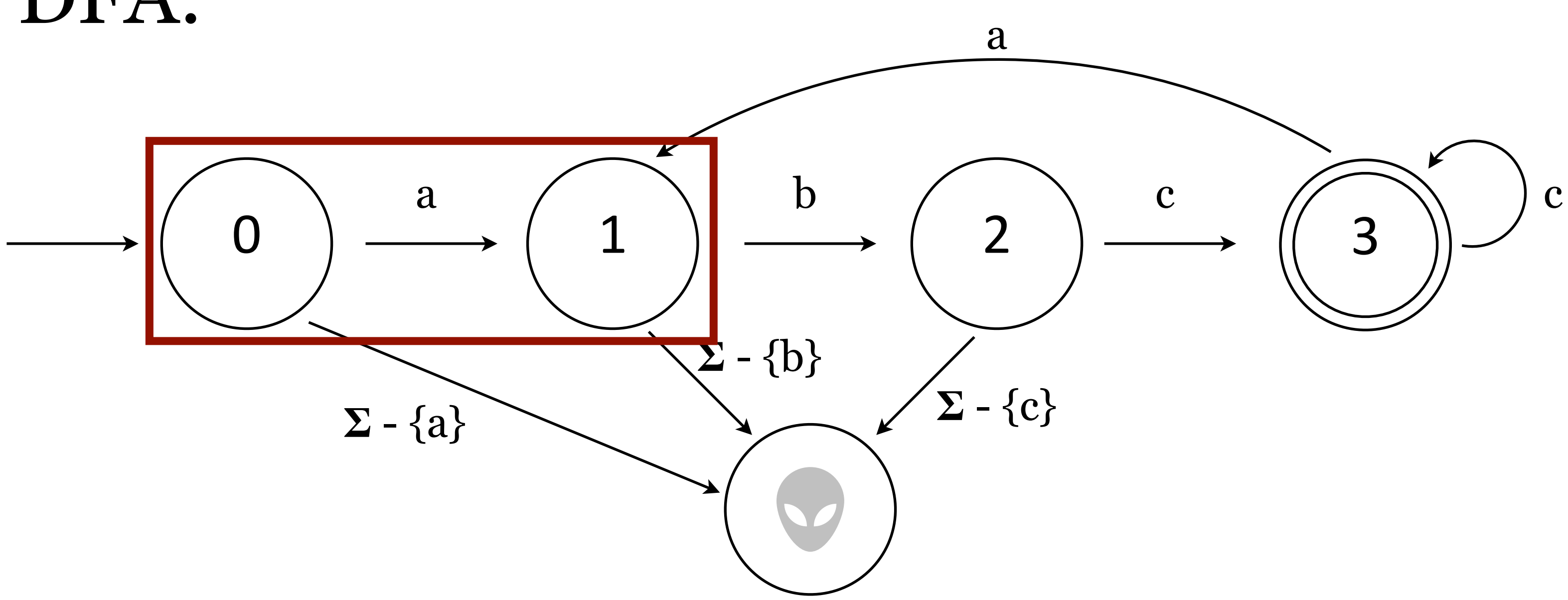
We can use a **transition table** to implement DFAs in our programs.

Deterministic Finite Automata

Let $\mathbf{\Sigma} = \{a, b, c\}$

RegEx: $(a\ b\ c^{+})^{+}$

DFA:



Transition Table

	a	b	c
0	1	-	-
1			
2			
3			

We can use a **transition table** to implement DFAs in our programs.

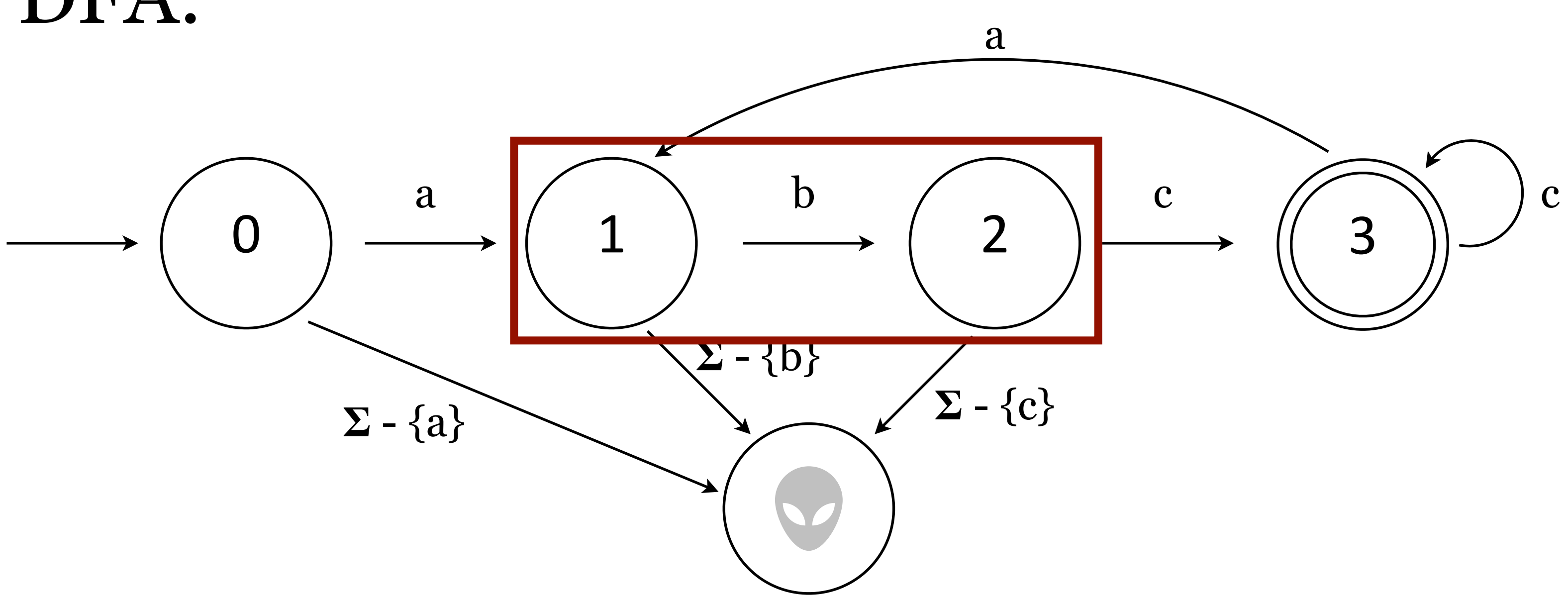
"If we're in state 0 and consume an 'a' from the input then we move to state 1."

Deterministic Finite Automata

Let $\mathbf{\Sigma} = \{a, b, c\}$

RegEx: $(a\ b\ c^{+})^{+}$

DFA:



Transition Table

	a	**b**	**c**
0	1	-	-
1	-	2	-
2			
3			

We can use a **transition table** to implement DFAs in our programs.

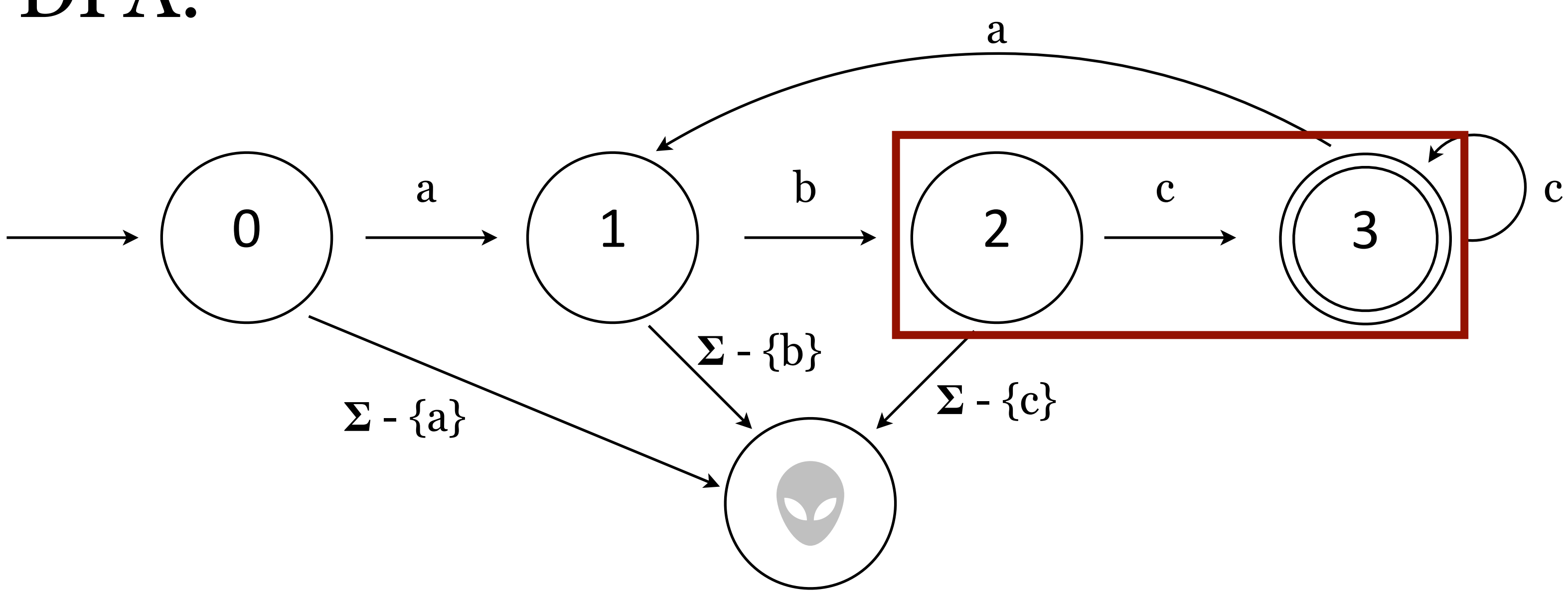
"If we're in state 1 and consume a 'b' from the input then we move to state 2."

Deterministic Finite Automata

Let $\mathbf{\Sigma} = \{a, b, c\}$

RegEx: $(a\ b\ c^{+})^{+}$

DFA:



Transition Table

	a	b	c
0	1	-	-
1	-	2	-
2	-	-	3
3			

We can use a **transition table** to implement DFAs in our programs.

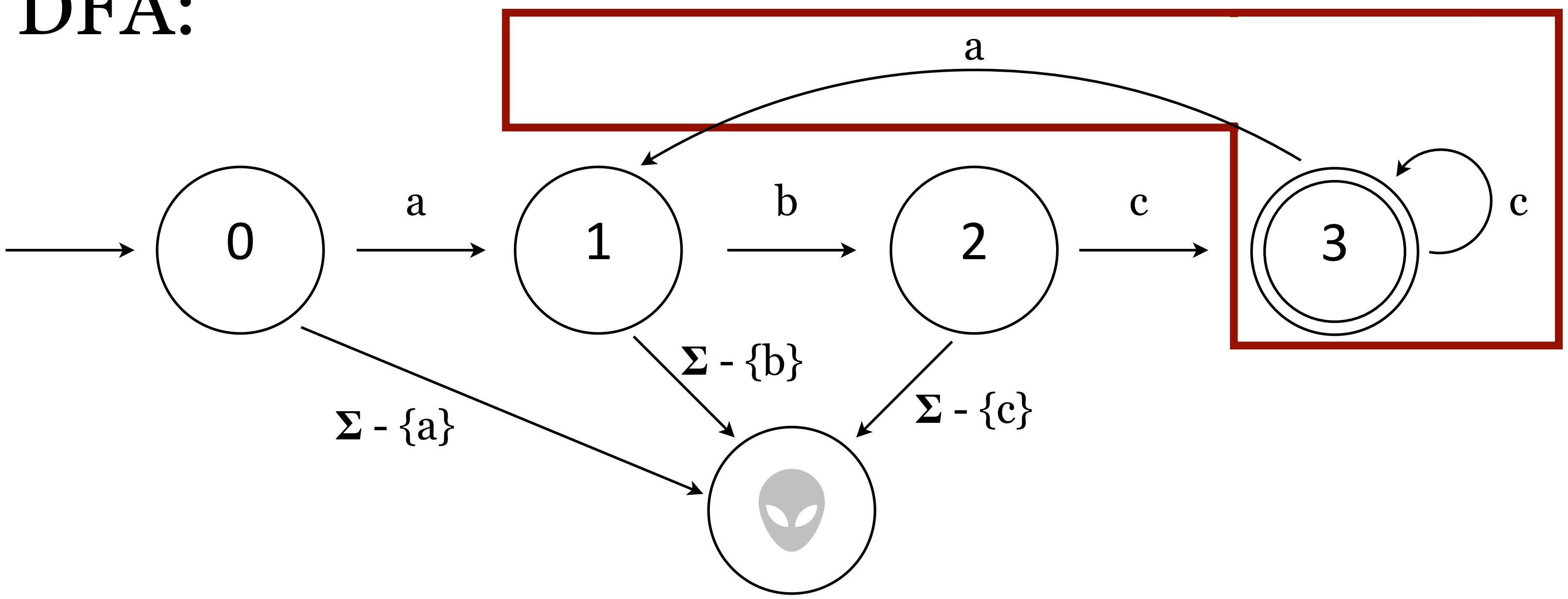
“If we’re in state 2 and consume a ‘c’ from the input then we move to state 3.”

Deterministic Finite Automata

Let $\mathbf{\Sigma} = \{a, b, c\}$

RegEx: $(a\ b\ c^{+})^{+}$

DFA:



Transition Table

	a	b	c
0	1	-	-
1	-	2	-
2	-	-	3
3	1	-	3

We can use a **transition table** to implement DFAs in our programs.

"If we're in state 3 and consume a 'c' from the input then we move to state 3. Else, if we're in state 3 and consume an 'a' from the input then we move to state 1."

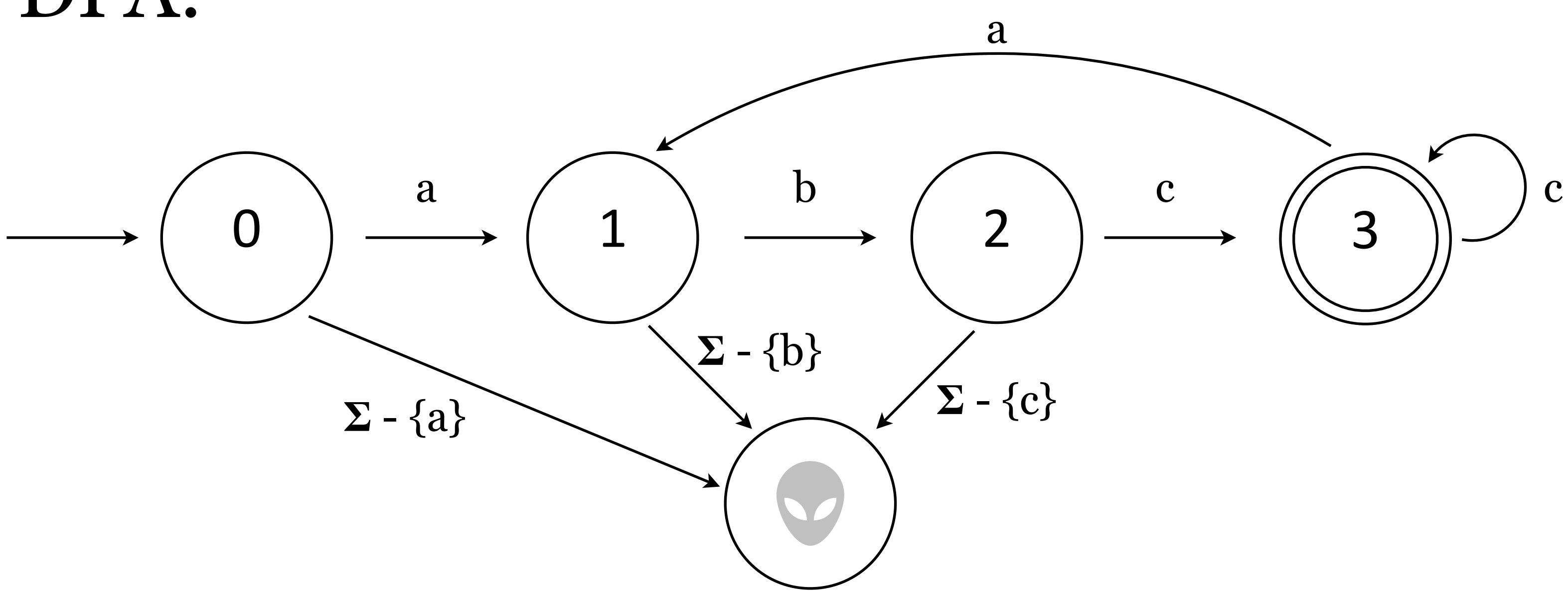
Also: mark state 3 as an accepting state.

Deterministic Finite Automata

Let $\boldsymbol{\Sigma} = \{a, b, c\}$

RegEx: $(a\ b\ c^{+})^{+}$

DFA:



Transition Table

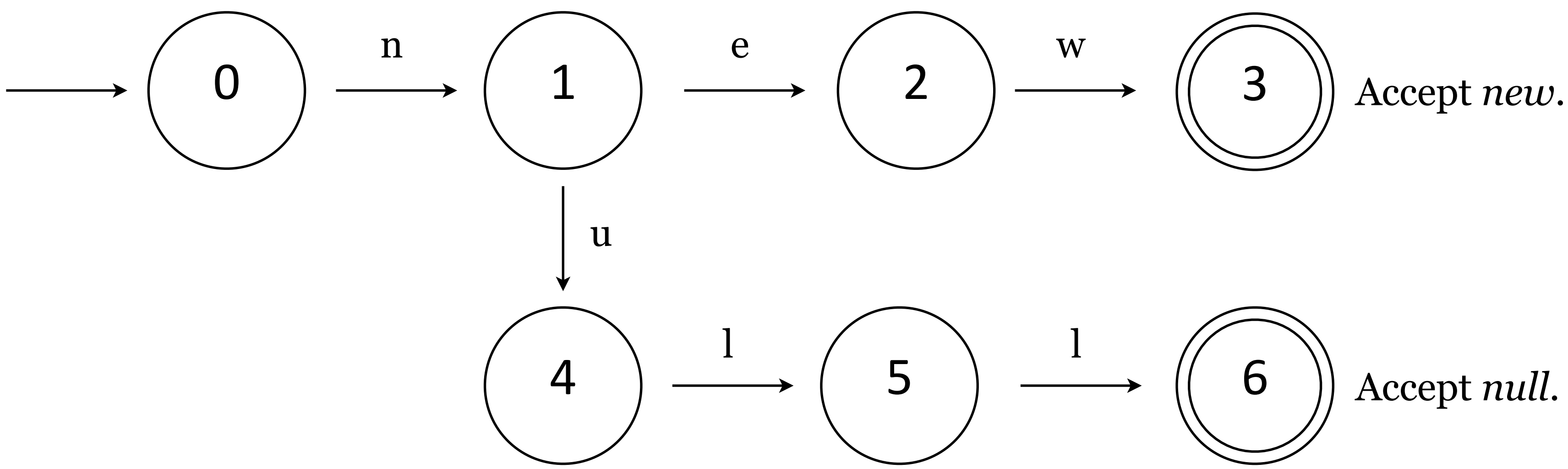
	a	**b**	**c**
0	1	-	-
1	-	2	-
2	-	-	3
$\boxed{\textbf{3}}$	1	-	3

We can use a **transition table** to implement DFAs in our programs.

Note:
Anything we can express in a RegEx we can write as a DFA.
Anything we can express in a DFA we can write as a RegEx.
So DFA == RegEx == DFA.

Deterministic Finite Automata

Example:
DFA for recognizing *new* and *null* tokens in a programming language.



Transition Table

	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z
0	-	-	-	-	-	-	-	-	-	-	-	-	-	1	-	-	-	-	-	-	-	-	-	-	-	-
1	-	-	-	-	2	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	4	-	-	-	-	-
2	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	3	-	-	-
3	accept *new*																									
4	-	-	-	-	-	-	-	-	-	-	-	5	-	-	-	-	-	-	-	-	-	-	-	-	-	-
5	-	-	-	-	-	-	-	-	-	-	-	6	-	-	-	-	-	-	-	-	-	-	-	-	-	-
6	accept *null*																									

Back to Pattern Matching to Make Tokens

Let's revisit our precise RegExes and make DFAs.

identifier

alphabetic followed by
0 or more alphanumerics

[a-z][a-z0-9]*

integer

0 or (digit from 1 — 9 followed by 0 or more digits from 0 — 9)

0|([1-9][0-9]*)

decimal

integer followed by '.'
followed by 1 or more integers

(0|([1-9][0-9]*)) '.' (0|([1-9][0-9]*))+

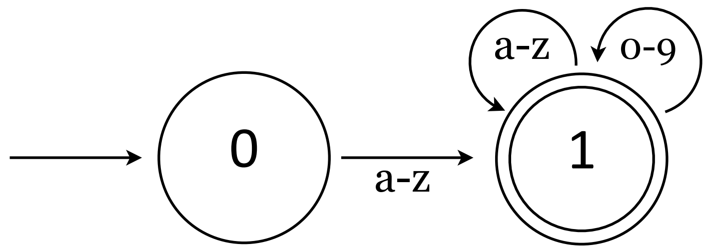
Back to Pattern Matching to Make Tokens

Let's revisit our precise RegExes and make DFAs.

identifier

alphabetic followed by
0 or more alphanumerics

[a-z][a-z0-9]*



integer

0 or (digit from 1 — 9 followed by 0 or more digits from 0 — 9)

0|([1-9][0-9]*)

decimal

integer followed by '.'
followed by 1 or more integers

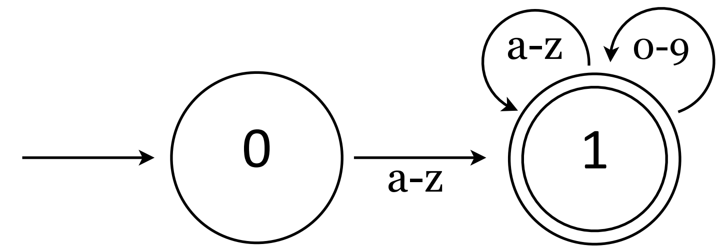
(0|([1-9][0-9]*)) '.' (0|([1-9][0-9]*))+

Back to Pattern Matching to Make Tokens

Let's revisit our precise RegExes and make DFAs.

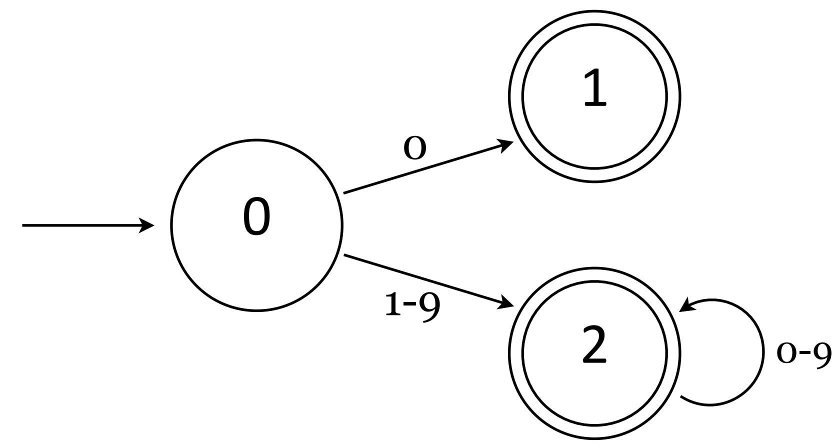
identifier
alphabetic followed by
0 or more alphanumerics

[a-z][a-z0-9]*



integer
0 or (digit from 1 — 9 followed by 0 or more digits from 0 — 9)

0|([1-9][0-9]*)



decimal
integer followed by '.'
followed by 1 or more integers

(0|([1-9][0-9]*)) '.' (0|([1-9][0-9]*))+

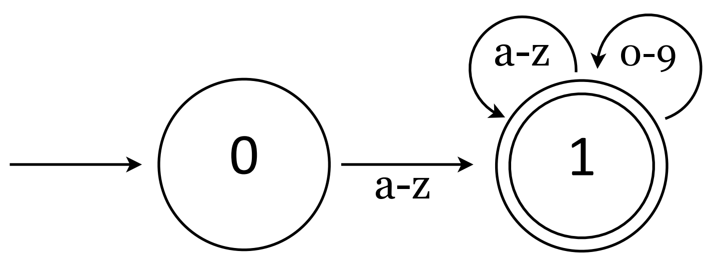
Back to Pattern Matching to Make Tokens

Let's revisit our precise RegExes and make DFAs.

identifier

alphabetic followed by
0 or more alphanumerics

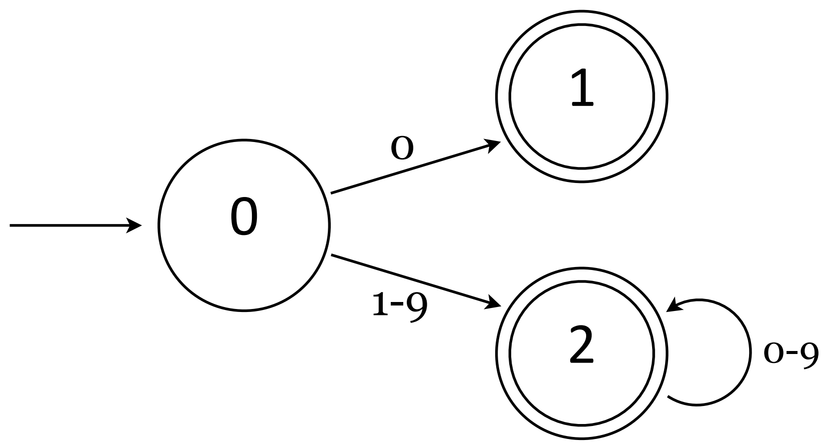
[a-z][a-z0-9]*



integer

0 or (digit from 1 — 9 followed by 0 or more digits from 0 — 9)

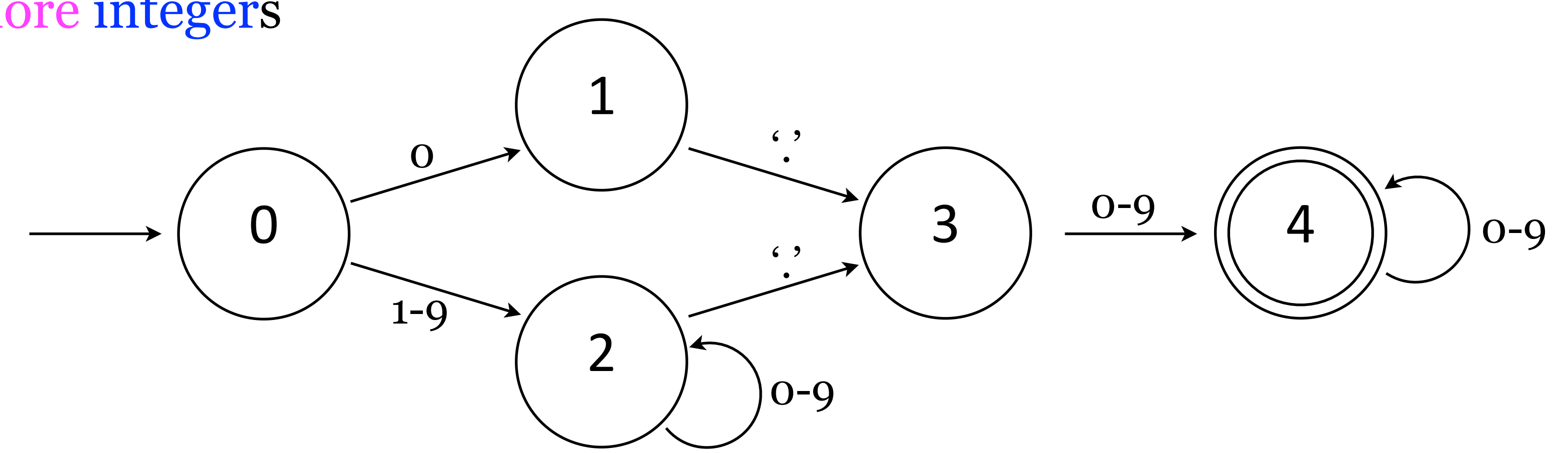
0|([1-9][0-9]*)



decimal

integer followed by '.'
followed by 1 or more integers

(0|([1-9][0-9]*)) '.' (0|([1-9][0-9]*))+



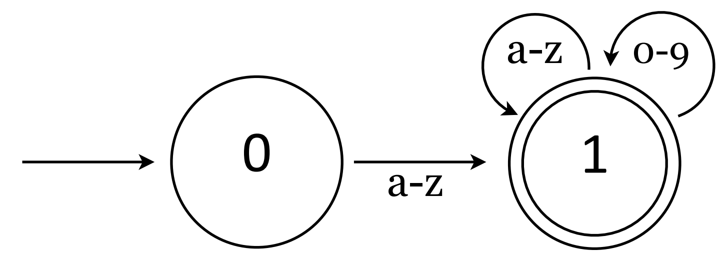
Back to Pattern Matching to Make Tokens

Let’s revisit our precise RegExes and make DFAs.

identifier

alphabetic followed by
0 or more alphanumerics

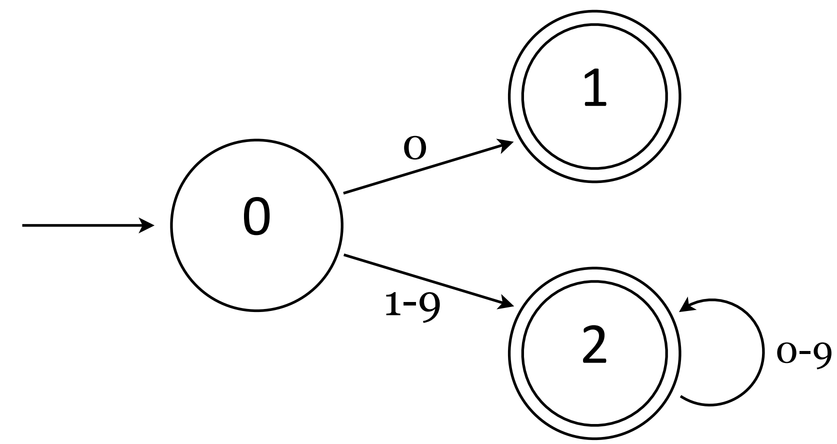
[a-z][a-z0-9]*



integer

0 or (digit from 1 — 9 followed by 0 or more digits from 0 — 9)

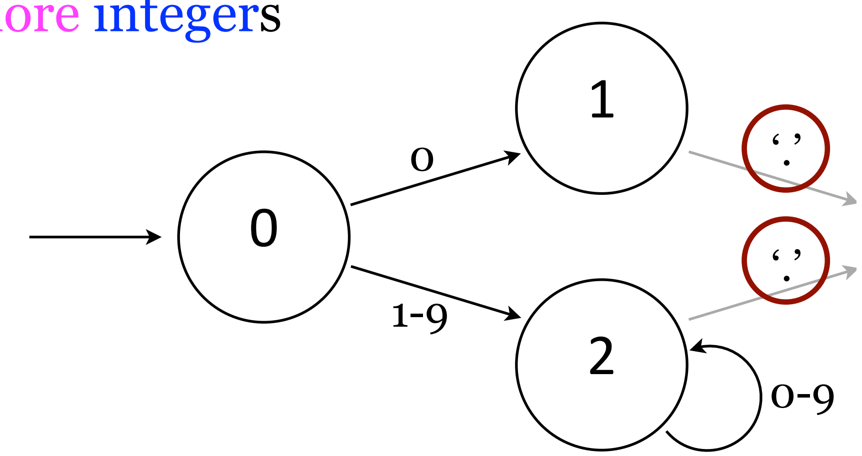
0|([1-9][0-9]*)



decimal

integer followed by ‘.’
followed by 1 or more integers

(0|([1-9][0-9]*))‘.’(0|([1-9][0-9]*))+



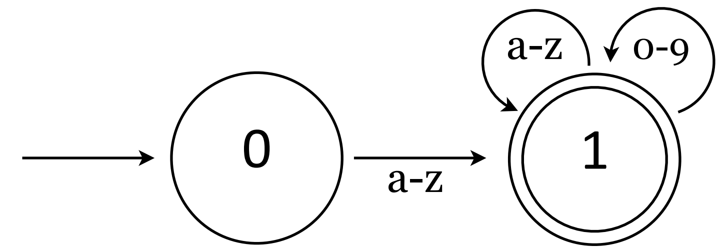
We use quotes here to disambiguate between a literal dot (which is what we want in this case) and the regular expression symbol for a wildcard.

Back to Pattern Matching to Make Tokens

Let's revisit our precise RegExes and make DFAs.

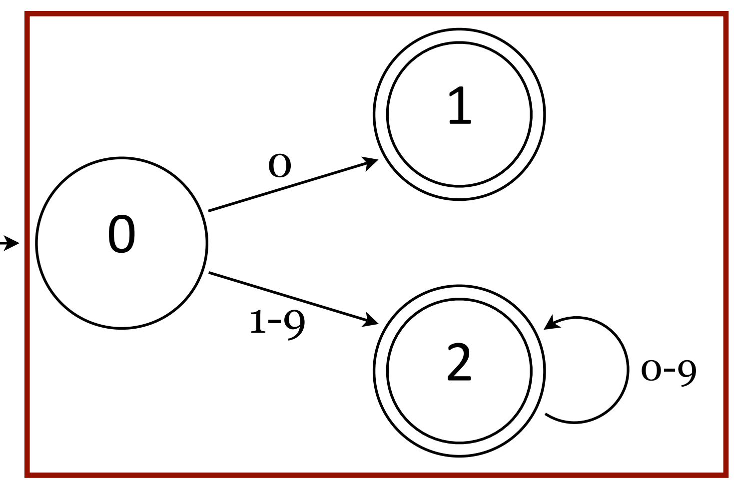
identifier
alphabetic followed by
0 or more alphanumerics

[a-z][a-z0-9]*



integer
0 or (digit from 1 — 9 followed by 0 or more digits from 0 — 9)

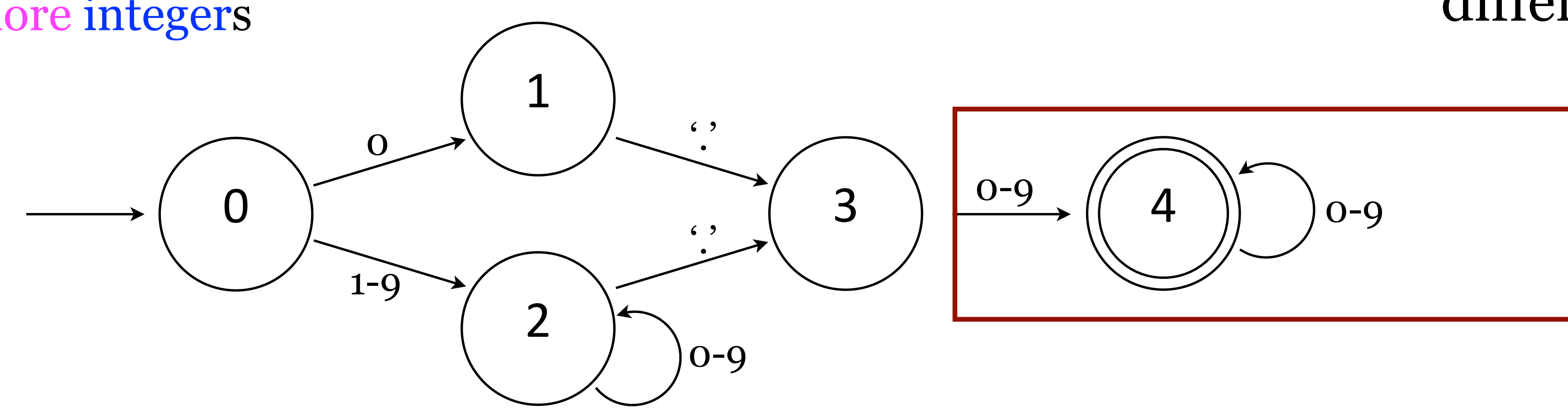
0|([1-9][0-9]*)



decimal
integer followed by '.'
followed by 1 or more integers

(0|([1-9][0-9]*)) '.' (0|([1-9][0-9]*))+

Why are these different?



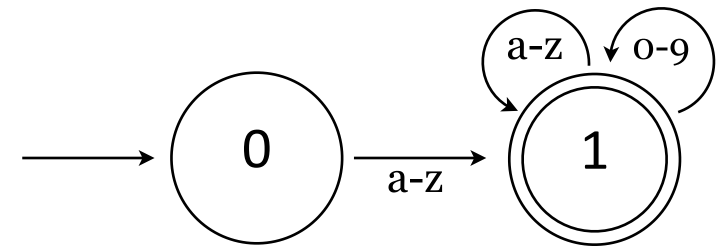
Back to Pattern Matching to Make Tokens

Let's revisit our precise RegExes and make DFAs.

identifier

alphabetic followed by
0 or more alphanumerics

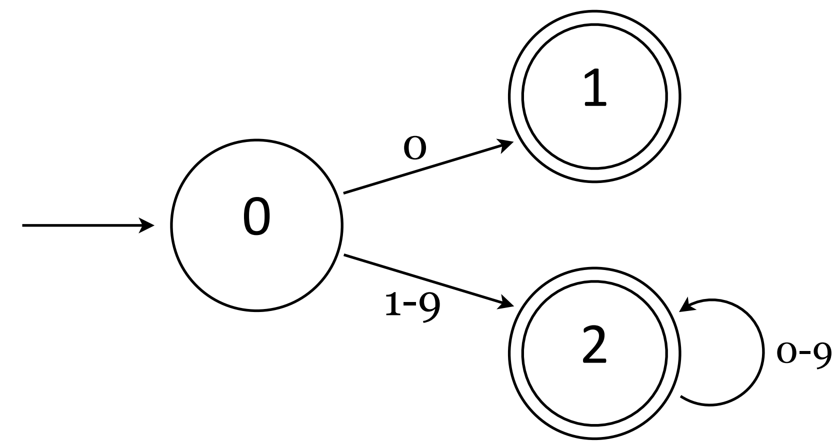
[a-z][a-z0-9]*



integer

0 or (digit from 1 — 9 followed by 0 or more digits from 0 — 9)

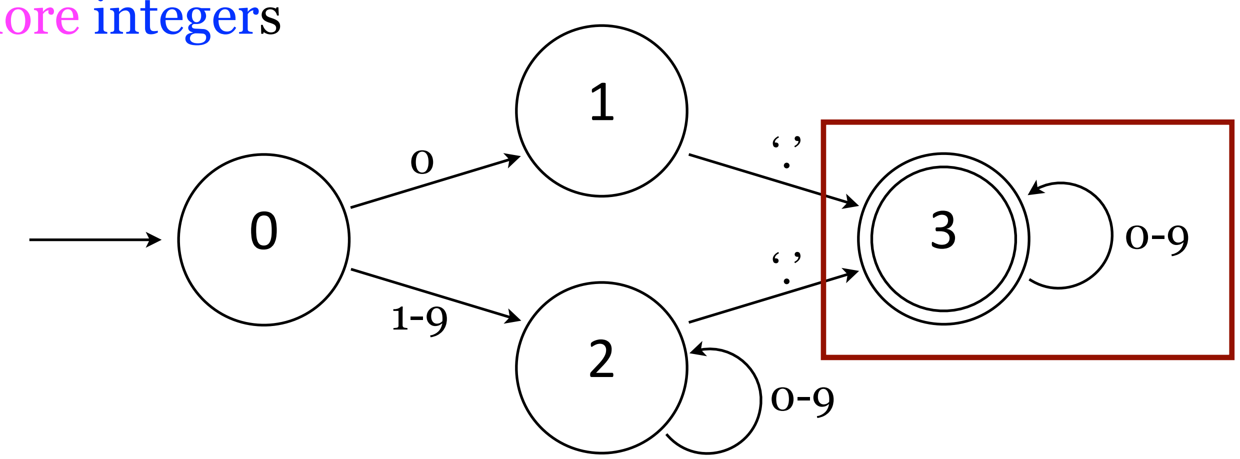
0|([1-9][0-9]*)



decimal

integer followed by '.'
followed by 1 or more integers

(0|([1-9][0-9]*)) '.' (0|([1-9][0-9]*))+



Why not do this?

Deterministic Finite Automata

Example: DFAs for recognizing a few common programming language constructs.

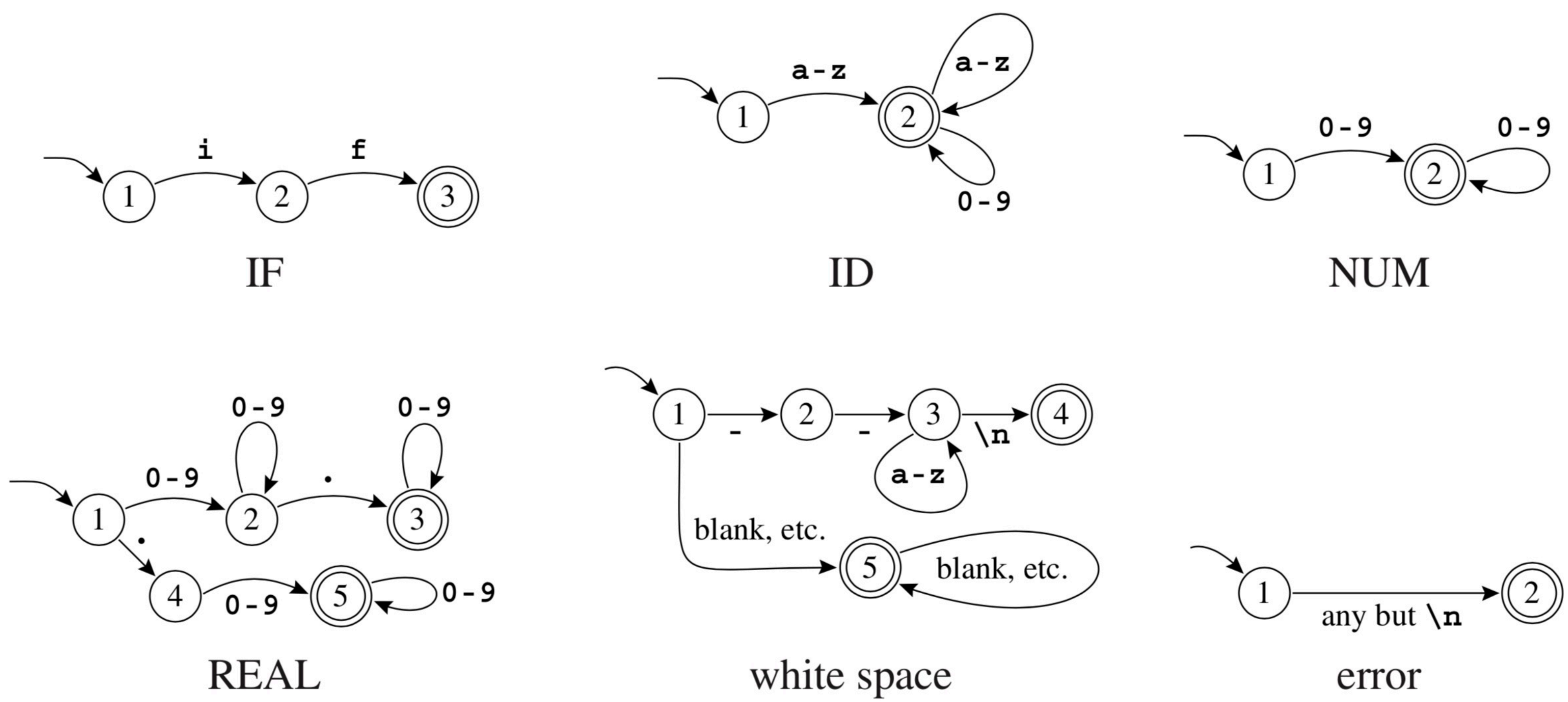


FIGURE 2.3. Finite automata for lexical tokens. The states are indicated by circles; final states are indicated by double circles. The start state has an arrow coming in from nowhere. An edge labeled with several characters is shorthand for many parallel edges.

From *Modern Compiler Implementation* by Andrew W. Appel.

Deterministic Finite Automata

Example: DFAs for recognizing a few common programming language constructs.

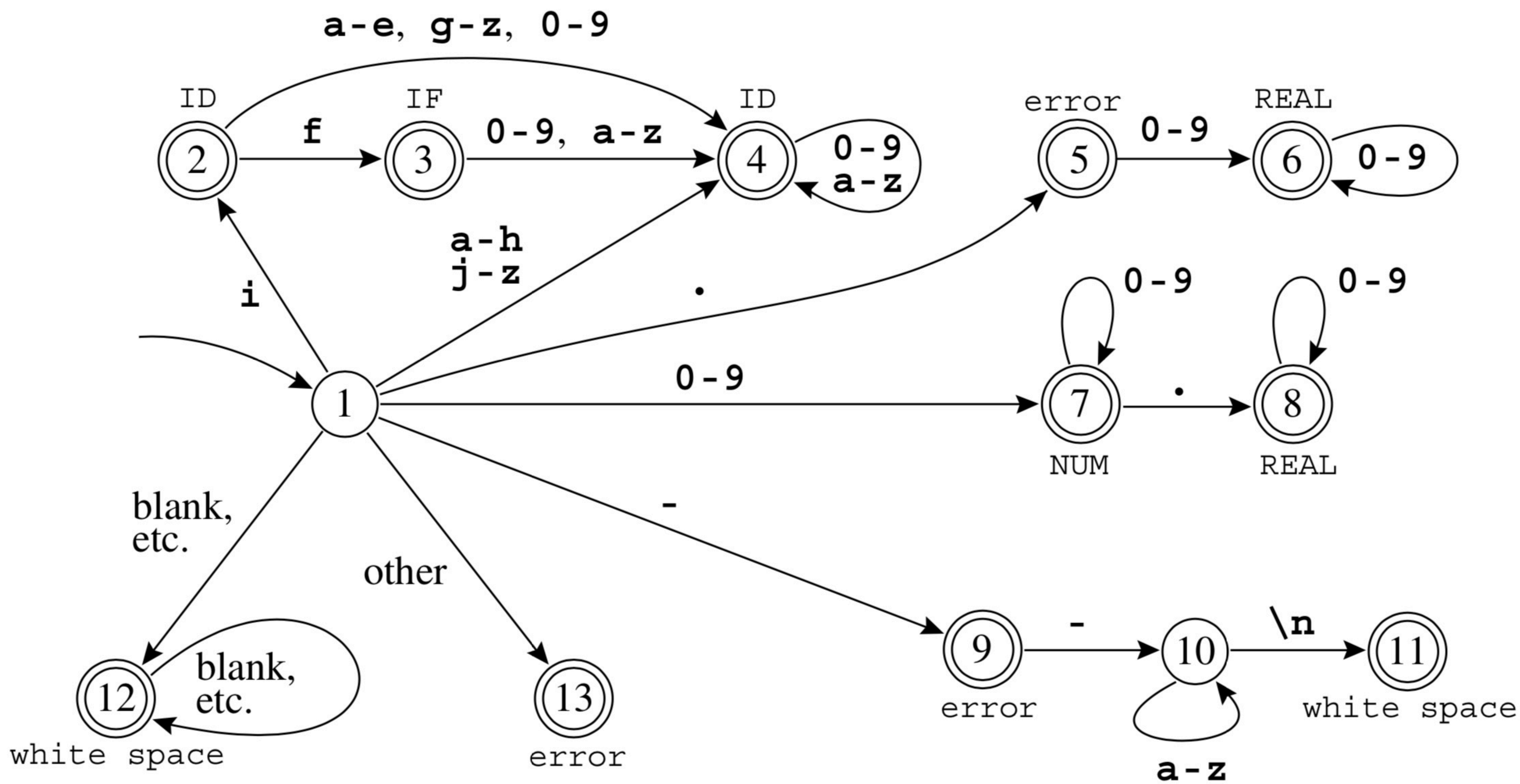


FIGURE 2.4. Combined finite automaton.

From *Modern Compiler Implementation* by Andrew W. Appel.

Deterministic Finite Automata

Example: DFAs for recognizing a few common programming language constructs.

We can encode this machine as a transition matrix: a two-dimensional array (a vector of vectors), subscripted by state number and input character. There will be a "dead" state (state 0) that loops to itself on all characters; we use this to encode the absence of an edge.

```
int edges[][] = {  /* ···0 1 2···-···e f g h i j··· */
/* state 0 */    {0,0,···0,0,0···0···0,0,0,0,0,0···},
/* state 1 */    {0,0,···7,7,7···9···4,4,4,4,2,4···},
/* state 2 */    {0,0,···4,4,4···0···4,3,4,4,4,4···},
/* state 3 */    {0,0,···4,4,4···0···4,4,4,4,4,4···},
/* state 4 */    {0,0,···4,4,4···0···4,4,4,4,4,4···},
/* state 5 */    {0,0,···6,6,6···0···0,0,0,0,0,0···},
/* state 6 */    {0,0,···6,6,6···0···0,0,0,0,0,0···},
/* state 7 */    {0,0,···7,7,7···0···0,0,0,0,0,0···},
/* state 8 */    {0,0,···8,8,8···0···0,0,0,0,0,0···},
    et cetera
}
```

There must also be a "finality" array, mapping state numbers to actions – final state 2 maps to action ID, and so on.

From *Modern Compiler Implementation* by Andrew W. Appel.

Deterministic Finite Automata

Example: DFAs for recognizing a few common programming language constructs.

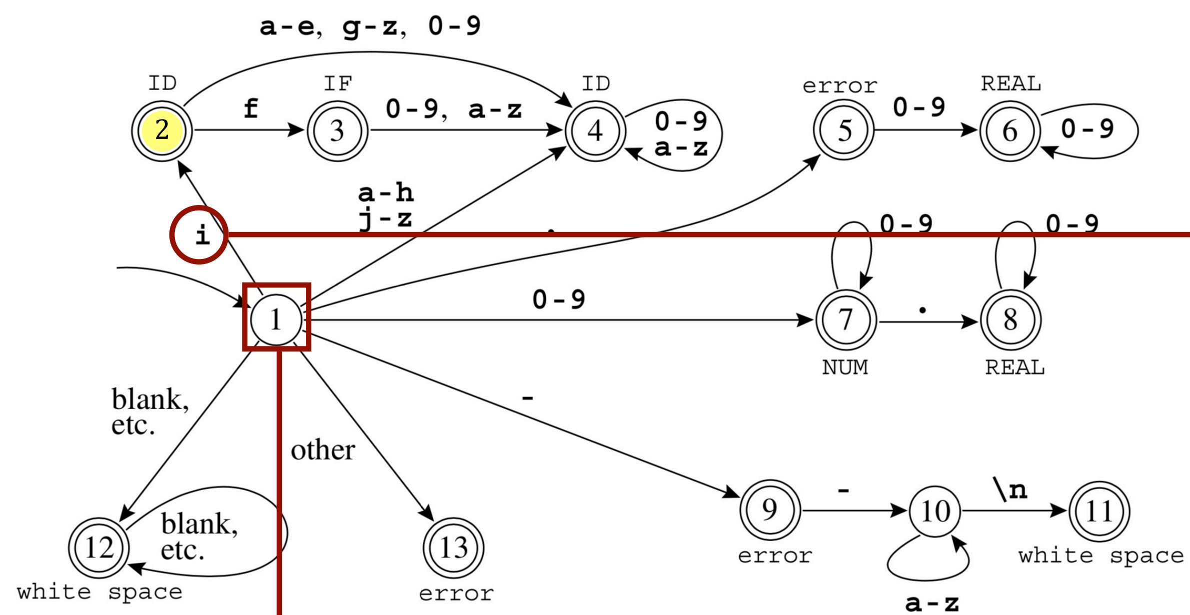


FIGURE 2.4. Combined finite automaton.

```
int edges[][] = {   /* ···0 1 2···-···e f g h i j··· */
/* state 0 */     {0,0,···0,0,0···0···0,0,0,0,0,0···},
/* state 1 */     {0,0,···7,7,7···9···4,4,4,4,2,4···},
/* state 2 */     {0,0,···4,4,4···0···4,3,4,4,4,4···},
/* state 3 */     {0,0,···4,4,4···0···4,4,4,4,4,4···},
/* state 4 */     {0,0,···4,4,4···0···4,4,4,4,4,4···},
/* state 5 */     {0,0,···6,6,6···0···0,0,0,0,0,0···},
/* state 6 */     {0,0,···6,6,6···0···0,0,0,0,0,0···},
/* state 7 */     {0,0,···7,7,7···0···0,0,0,0,0,0···},
/* state 8 */     {0,0,···8,8,8···0···0,0,0,0,0,0···},
    et cetera
}
```

Deterministic Finite Automata

Example: DFAs for recognizing a few common programming language constructs.

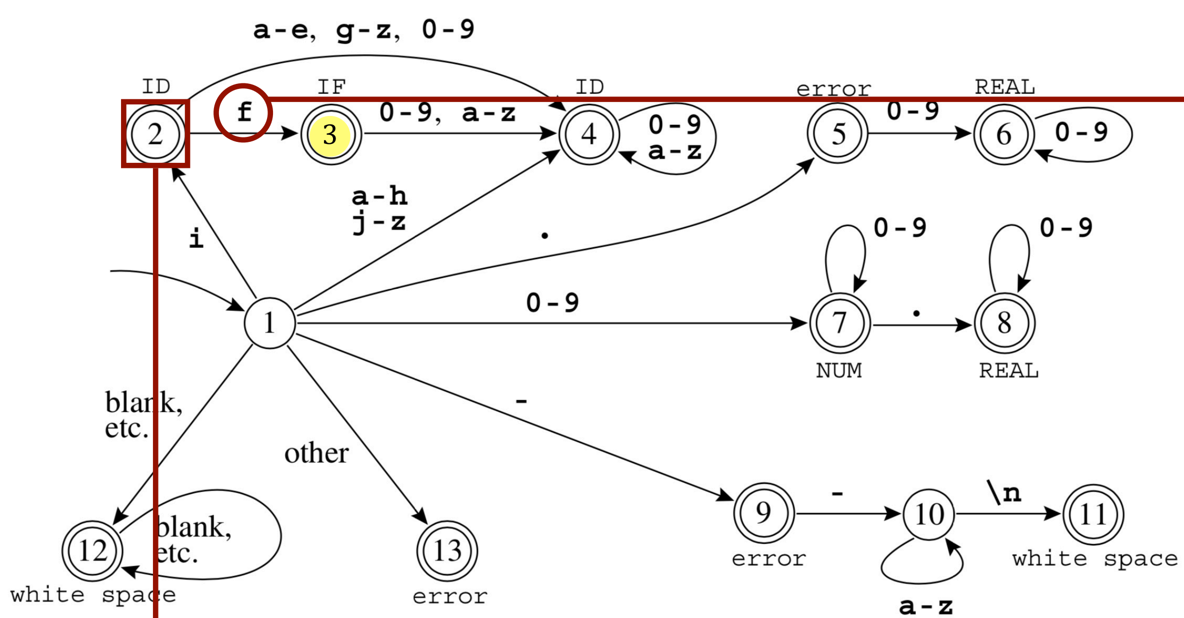


FIGURE 2.4. Combined finite automaton.

```
int edges[][] = {   /* ···0 1 2···-···e f g h i j··· */
/* state 0 */     {0,0,···0,0,0···0···0 0 0,0,0,0···},
/* state 1 */     {0,0,···7,7,7···9···4 4 4,4,2,4···},
/* state 2 */     {0,0,···4,4,4···0···4 3 4,4,4,4···},
/* state 3 */     {0,0,···4,4,4···0···4 4 4,4,4,4···},
/* state 4 */     {0,0,···4,4,4···0···4 4 4,4,4,4···},
/* state 5 */     {0,0,···6,6,6···0···0 0 0,0,0,0···},
/* state 6 */     {0,0,···6,6,6···0···0 0 0,0,0,0···},
/* state 7 */     {0,0,···7,7,7···0···0 0 0,0,0,0···},
/* state 8 */     {0,0,···8,8,8···0···0 0 0,0,0,0···},
    et cetera
}
```

Deterministic Finite Automata

DFAs have

- a set of input symbols
- a set of states
- a denoted start state
- a transitions to move from one state to another
- one or more denoted accepting states

where…

- no state can transition on an empty string
- for each state *s* and input symbol *a*, there is only one edge labeled *a* leaving *s*. I.e., all transitions from one state to another are unique and unambiguous.

A DFA accepts input *x* iff there exists a **unique** path from the start state to one of the accepting states such that the labels along the edges of that path spell *x*.

This is what we have so far.
And this is what you need to implement Lex.

But… what if we have a finite automata where the paths are **not** unique, and there **could** be more than one similarly-labeled transitions from one state to another? What would happen then?

Nondeterministic Finite Automata

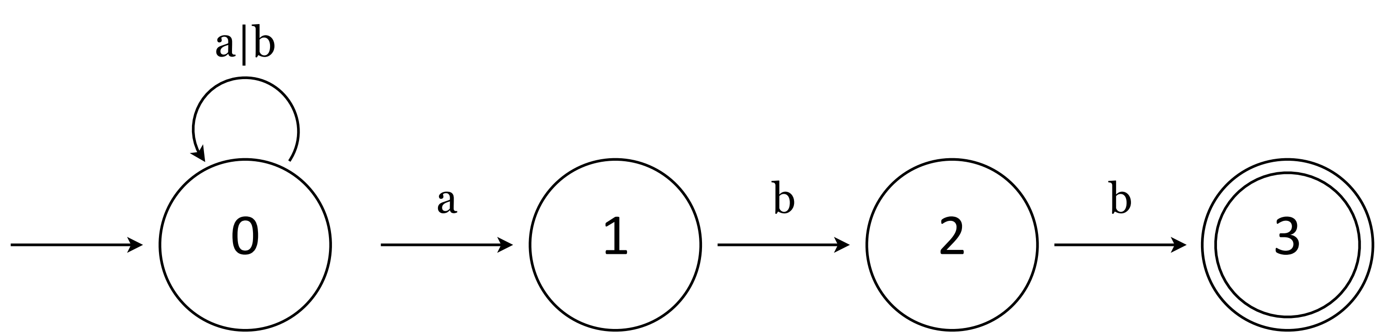
NFAs have

- a set of input symbols
- a set of states
- a denoted start state
- a transitions to move from one state to another
- one or more denoted accepting states

where…

- states **can** transition on an empty string
- for each state *s* and input symbol *a*, **could be many** edges labeled *a* leaving *s*. I.e., all transitions from one state to another are **not** unique and quite possibly ambiguous.

An NFA accepts input *x* iff there exists a ~~unique~~ path from the start state to one of the accepting states such that the labels along the edges of that path spell *x*.



What is the equivalent RegEx?

Nondeterministic Finite Automata

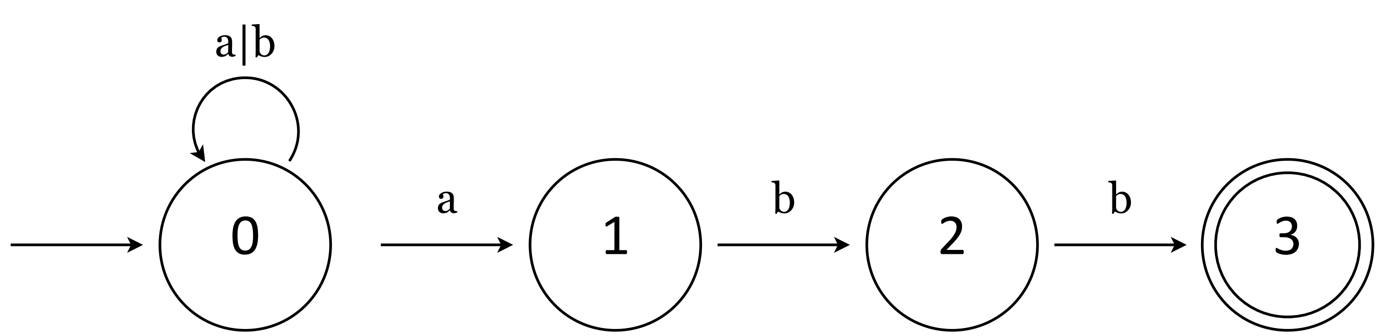
NFAs have

- a set of input symbols
- a set of states
- a denoted start state
- a transitions to move from one state to another
- one or more denoted accepting states

where…

- states **can** transition on an empty string
- for each state *s* and input symbol *a*, **could be many** edges labeled *a* leaving *s*. I.e., all transitions from one state to another are **not** unique and quite possibly ambiguous.

An NFA accepts input *x* iff there exists a ~~unique~~ path from the start state to one of the accepting states such that the labels along the edges of that path spell *x*.



$(a|b)^*abb$

Nondeterministic Finite Automata

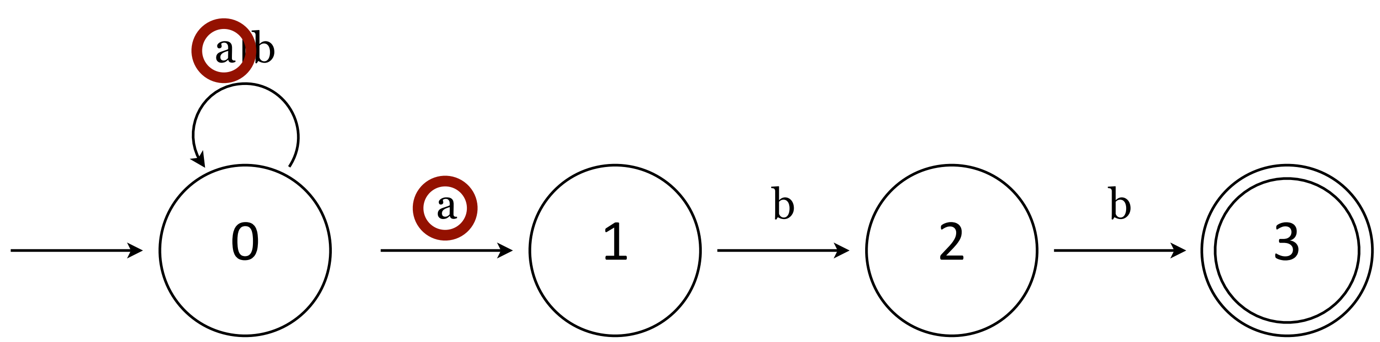
NFAs have

- a set of input symbols
- a set of states
- a denoted start state
- a transitions to move from one state to another
- one or more denoted accepting states

where…

- states **can** transition on an empty string
- for each state s and input symbol a, **could be many** edges labeled a leaving s. I.e., all transitions from one state to another are **not** unique and quite possibly ambiguous.

An NFA accepts input x iff there exists a ~~unique~~ path from the start state to one of the accepting states such that the labels along the edges of that path spell x.

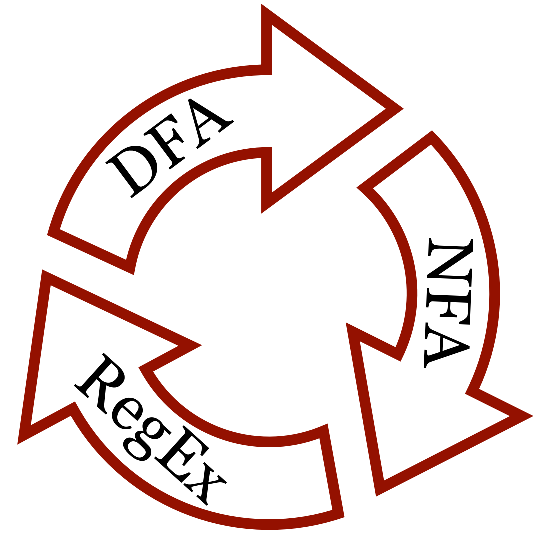


$(a|b)^*abb$

NFA = RegEx = DFA

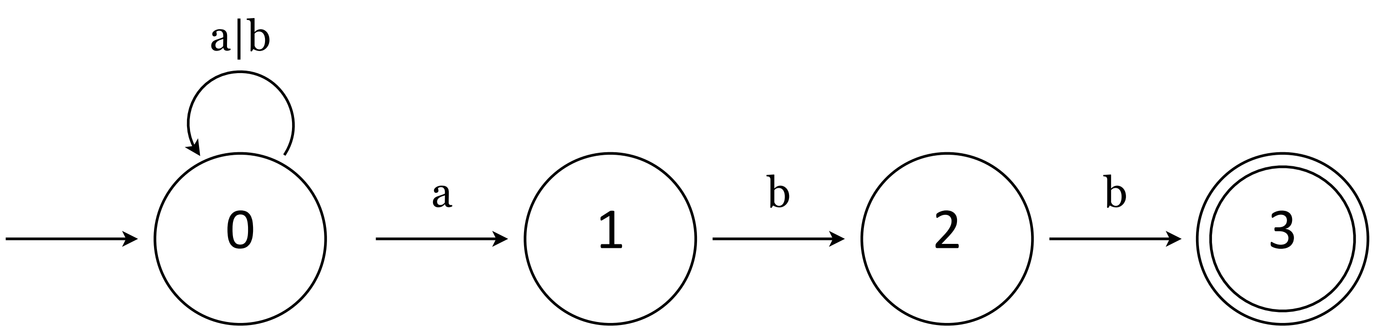
NFAs and RegExes and DFAs are equivalent

- DFA ↔ RegEx
- RegEx ↔ NFA
- ∴ DFA ↔ RegEx ↔ NFA



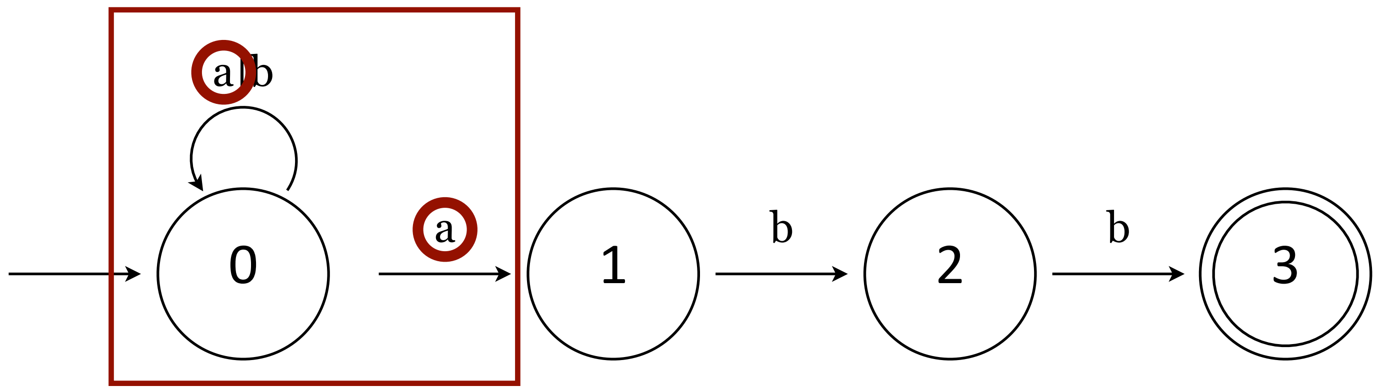
Also

- DFAs are a subset of NFAs
- Any NFA can be converted to a DFA by simulating sets of simultaneous states.
  - Each DFA state corresponds to a set of NFA states.
  - This is called *subset construction*, and it's fun.
  - There **could** be a lot of these. Possible exponential blowup. (2^n)



$(a|b)^*abb$

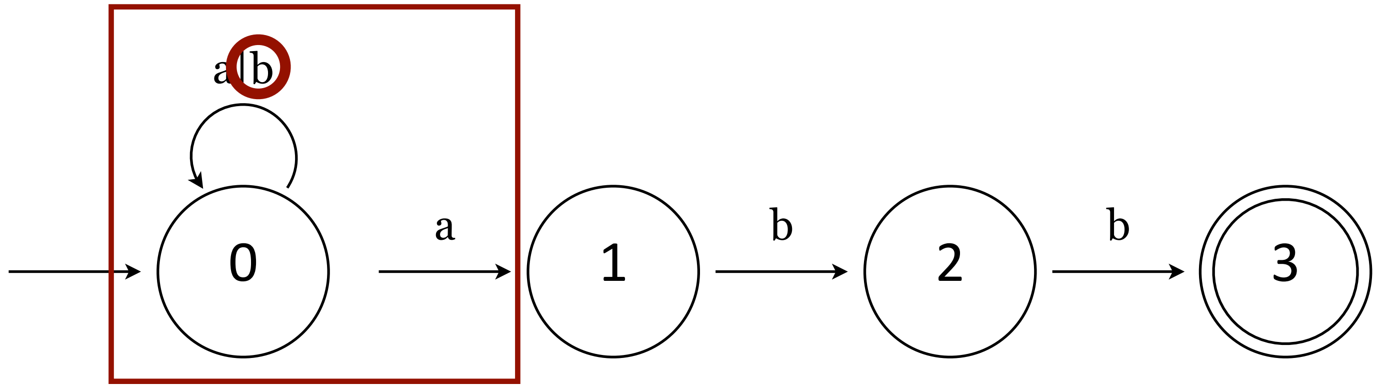
NFA to DFA via Subset Construction



	a	b
{0}	{0,1}	

In state 0, if we get an *a* then we can go to state 0 or 1.

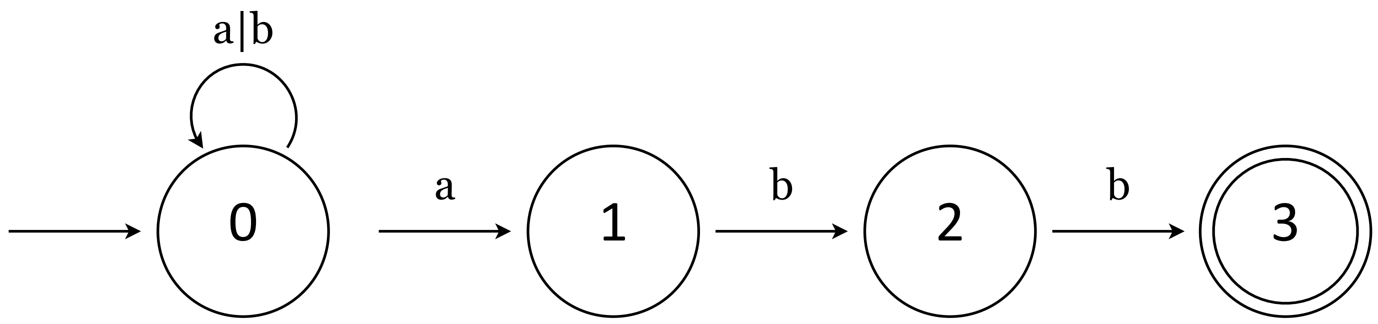
NFA to DFA via Subset Construction



	a	b
{0}	{0,1}	{0}

In state 0, if we get a *b* then we can go to state 0.

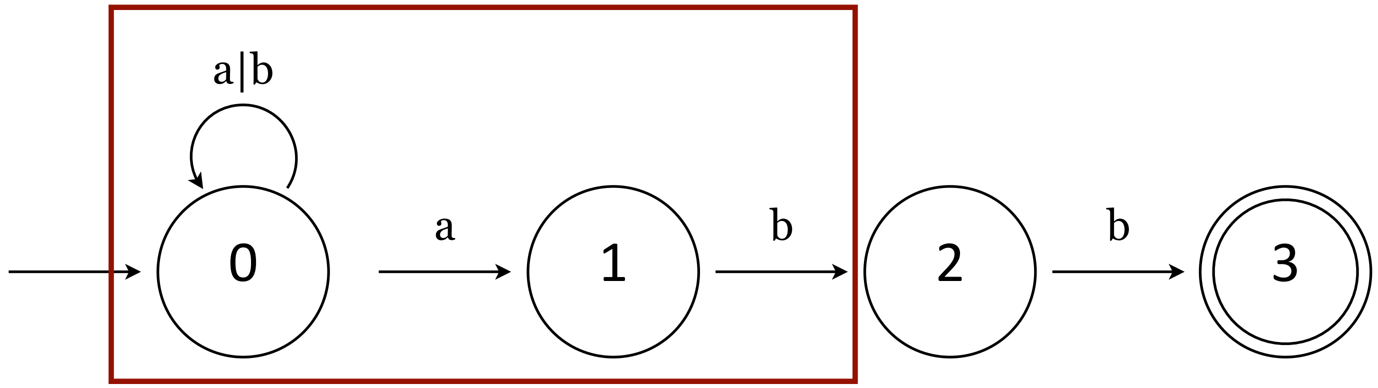
NFA to DFA via Subset Construction



	a	b
{0}	{0,1}	{0}
{0,1}		

We're now considering two possible sets of states in our transition table: {0} and {0,1}. But we've only enumerated the possibilities for {0}, so we add the newly introduced state to the table.

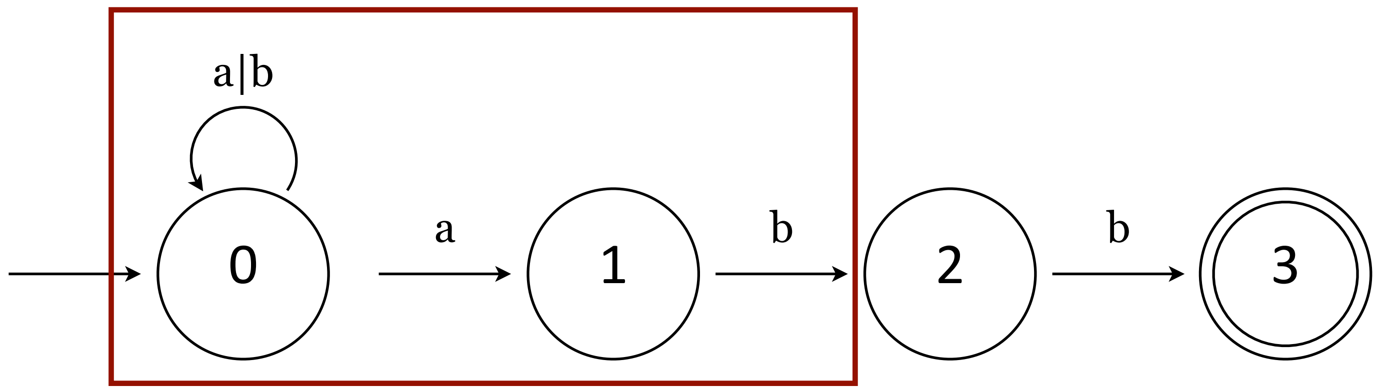
NFA to DFA via Subset Construction



	a	b
{0}	{0,1}	{0}
{0,1}		

Considering state {0,1} means analyzing what can happen if we are in state 0 and recording that in the transition table, then analyzing what can happen if we are in state 1 and recording that in the transition table, building sets of possibilities along the way.

NFA to DFA via Subset Construction

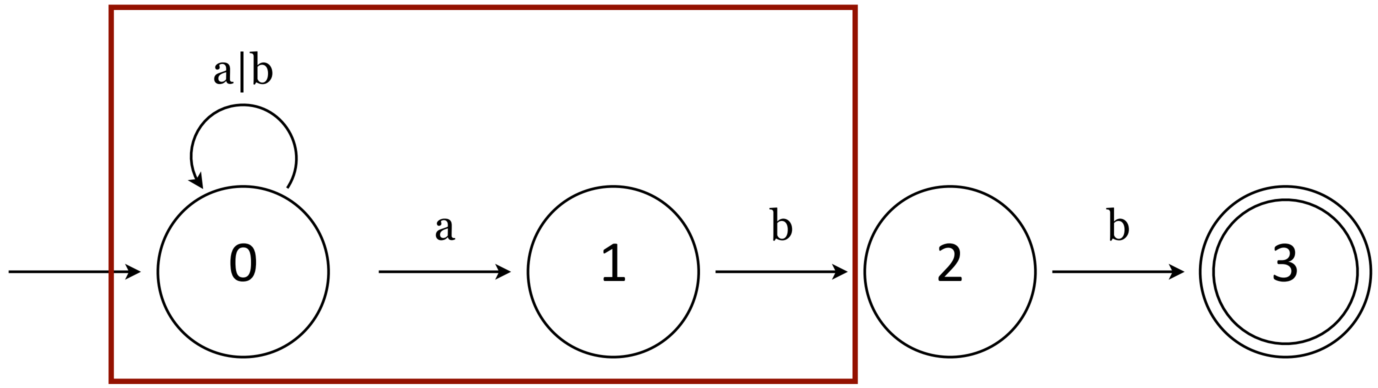


	a	b
{0}	{0,1}	{0}
{0,1}	{0,1}	

Considering state {0,1} means analyzing what can happen if we are in state 0 and recording that in the transition table, then analyzing what can happen if we are in state 1 and recording that in the transition table, building sets of possibilities along the way.

In state 0, if we get an *a* then we can go to state 0 or 1.
In state 1, we cannot get an *a* so there are no other options.

NFA to DFA via Subset Construction

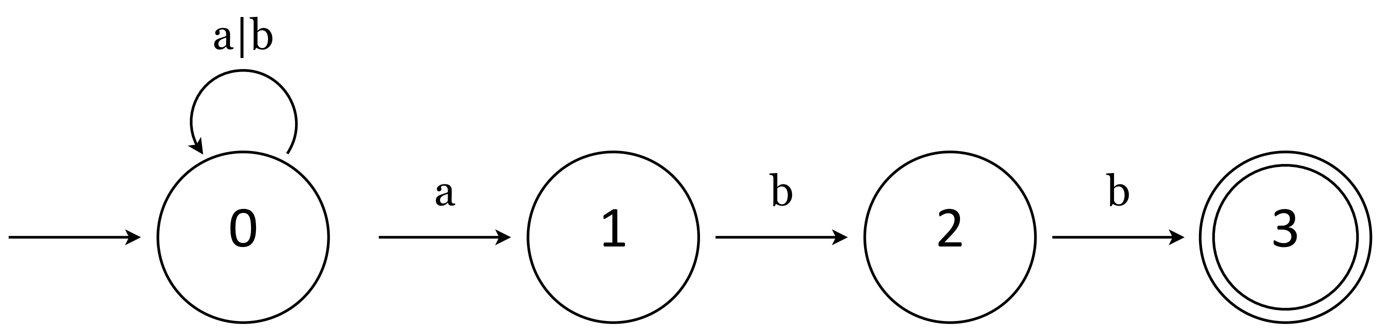


	a	**b**
{0}	{0,1}	{0}
{0,1}	{0,1}	{0,2}

Considering state {0,1} means analyzing what can happen if we are in state 0 and recording that in the transition table, then analyzing what can happen if we are in state 1 and recording that in the transition table, building sets of possibilities along the way.

In state 0, if we get a *b* then we can go to state 0.
In state 1, if we get a *b* then we can go to state 2.

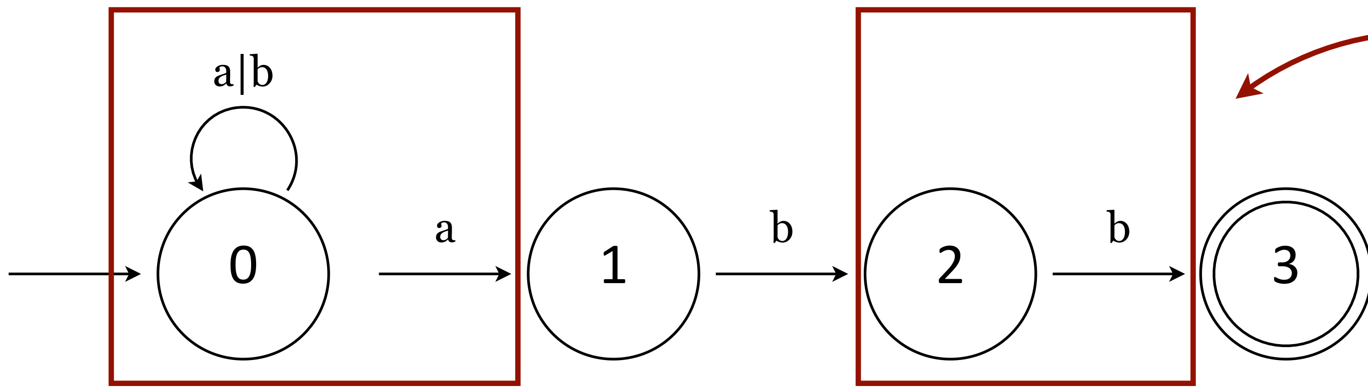
NFA to DFA via Subset Construction



	a	b
{0}	{0,1}	{0}
{0,1}	{0,1}	{0,2}
{0,2}		

We're now considering three possible sets of states in our transition table: {0}, {0,1}, and {0,2}. But we've only enumerated the possibilities for {0} and {0,1}, so we add the newly introduced state to the table.

NFA to DFA via Subset Construction

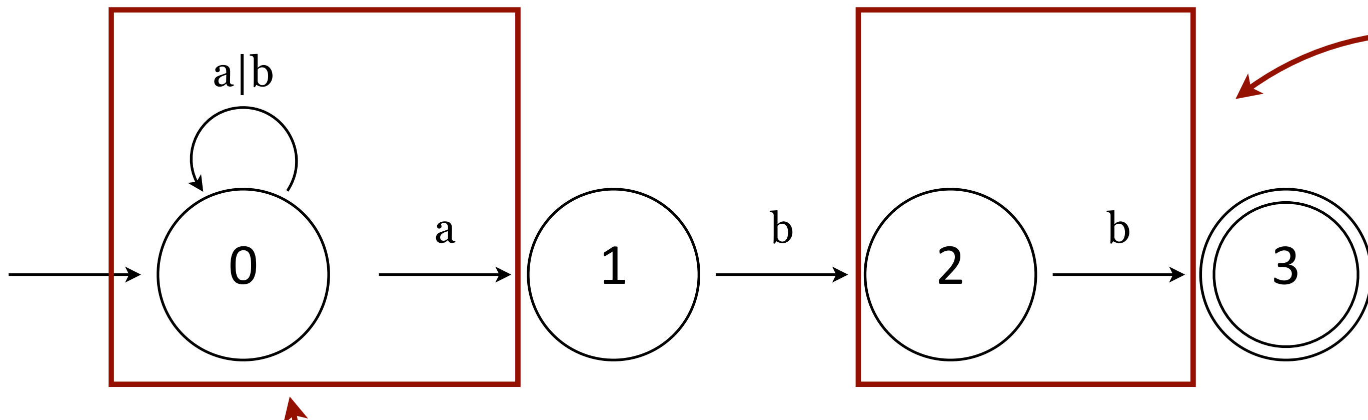


	a	b
{0}	{0,1}	{0}
{0,1}	{0,1}	{0,2}
{0,2}	{0,1}	

Considering state {0,2} means analyzing what can happen if we are in state 0 and recording that in the transition table, then analyzing what can happen if we are in state 2 and recording that in the transition table, building sets of possibilities along the way.

In state 0, if we get an *a* then we can go to state 0 or 1.
In state 2, we cannot get an *a* so there are no other options.

NFA to DFA via Subset Construction

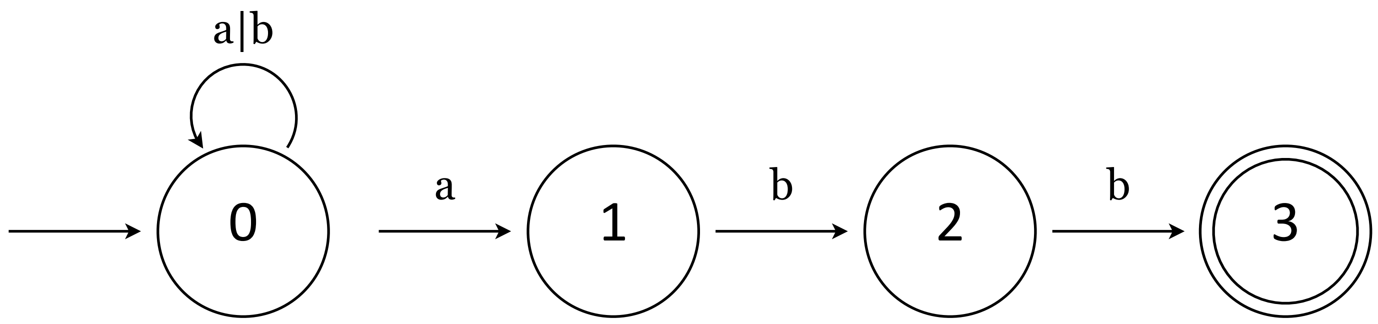


	a	b
{0}	{0,1}	{0}
{0,1}	{0,1}	{0,2}
{0,2}	{0,1}	{0,3}

Considering state {0,2} means analyzing what can happen if we are in state 0 and recording that in the transition table, then analyzing what can happen if we are in state 2 and recording that in the transition table, building sets of possibilities along the way.

In state 0, if we get a *b* then we can go to state 0.
In state 2, if we get a *b* then we can go to state 3.

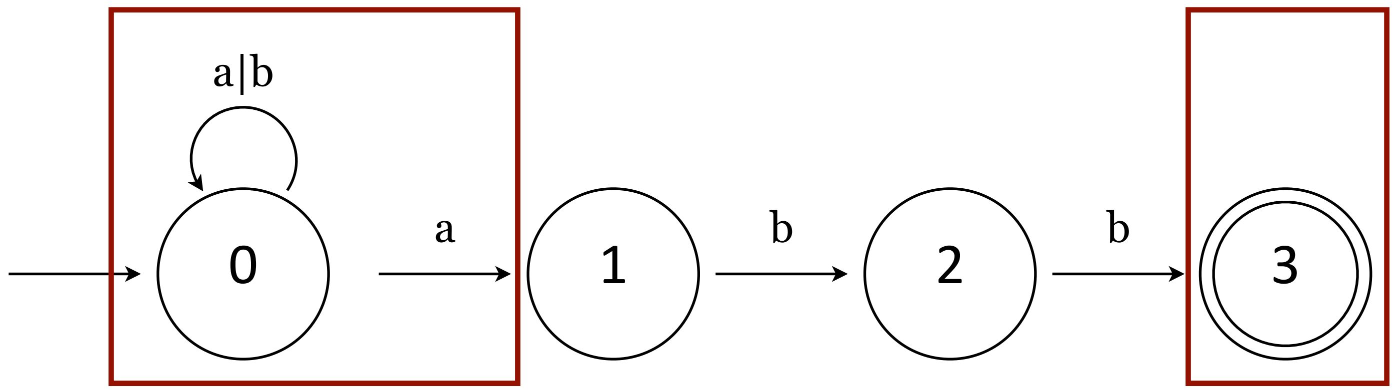
NFA to DFA via Subset Construction



	a	b
{0}	{0,1}	{0}
{0,1}	{0,1}	{0,2}
{0,2}	{0,1}	{0,3}
{0,3}		

Add the newly introduced state to the table.

NFA to DFA via Subset Construction

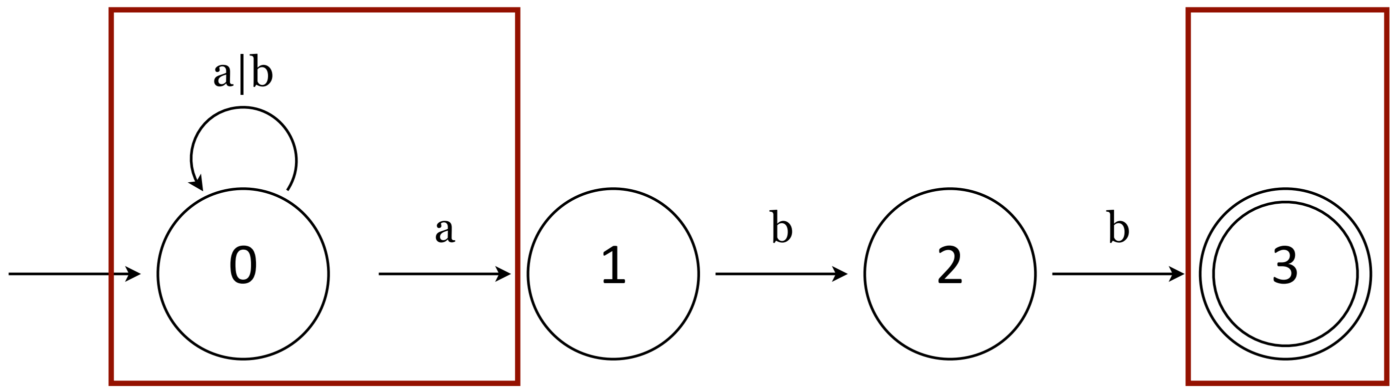


	a	b
{0}	{0,1}	{0}
{0,1}	{0,1}	{0,2}
{0,2}	{0,1}	{0,3}
{0,3}	{0,1}	

Considering state {0,3} means analyzing what can happen if we are in state 0 and recording that in the transition table, then analyzing what can happen if we are in state 3 and recording that in the transition table, building sets of possibilities along the way.

In state 0, if we get an *a* then we can go to state 0 or 1.
In state 3, we cannot get an *a* so there are no other options.

NFA to DFA via Subset Construction

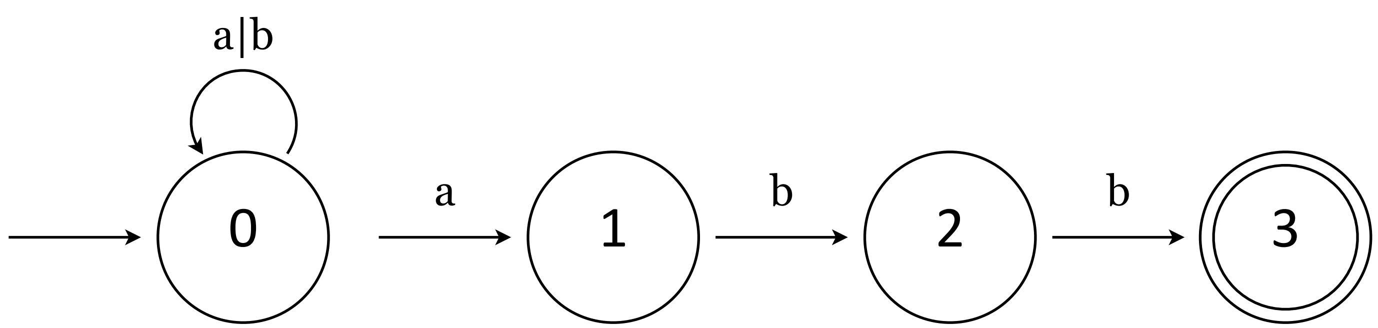


	a	b
{0}	{0,1}	{0}
{0,1}	{0,1}	{0,2}
{0,2}	{0,1}	{0,3}
{0,3}	{0,1}	{0}

Considering state {0,3} means analyzing what can happen if we are in state 0 and recording that in the transition table, then analyzing what can happen if we are in state 3 and recording that in the transition table, building sets of possibilities along the way.

In state 0, if we get a *b* then we can go to state 0.
In state 3, we cannot get an *b* so there are no other options.

NFA to DFA via Subset Construction

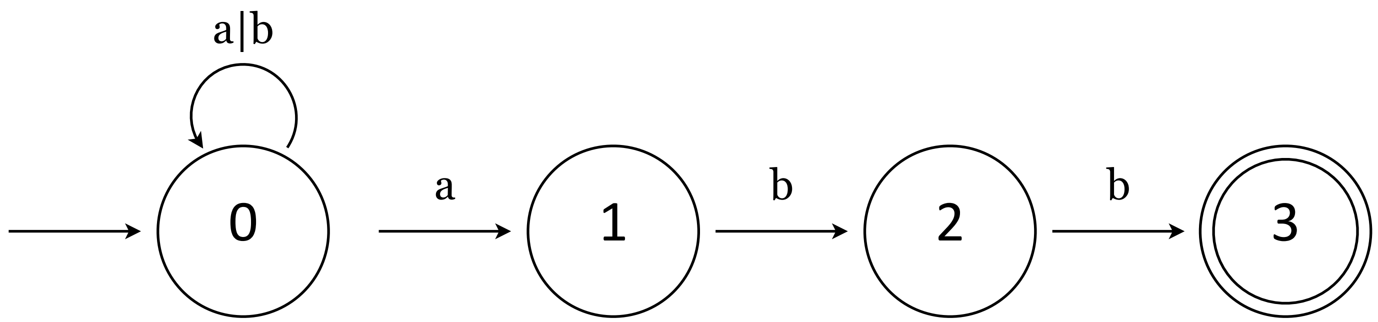


	a	b
{0}	{0,1}	{0}
{0,1}	{0,1}	{0,2}
{0,2}	{0,1}	{0,3}
{0,3}	{0,1}	{0}

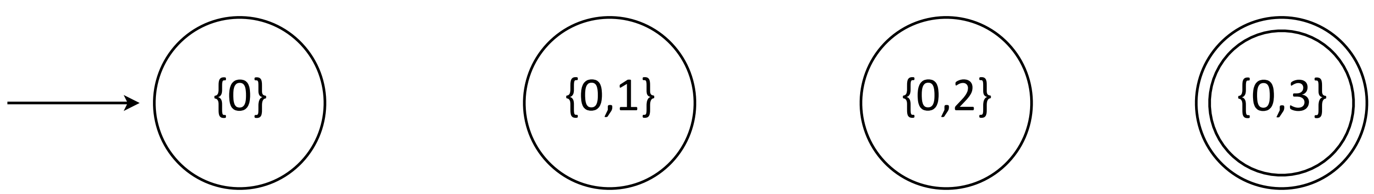
There are no new states to add to our transition table, so we're done constructing the subsets. (Also, the final state is an accepting state.)

Now we can take the transition table made from this NFA and use it to build a DFA by creating states labeled according to the transition table values.

NFA to DFA via Subset Construction

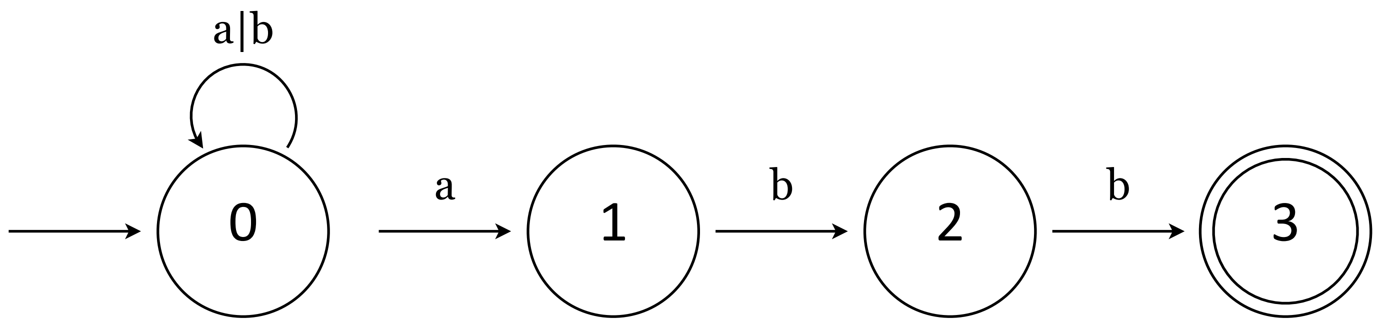


	a	b
{0}	{0,1}	{0}
{0,1}	{0,1}	{0,2}
{0,2}	{0,1}	{0,3}
{0,3}	{0,1}	{0}

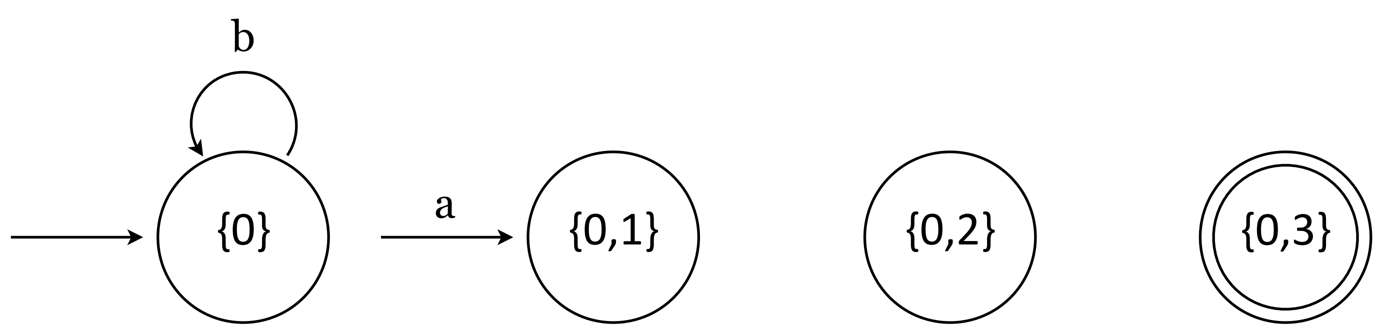


DFA

NFA to DFA via Subset Construction

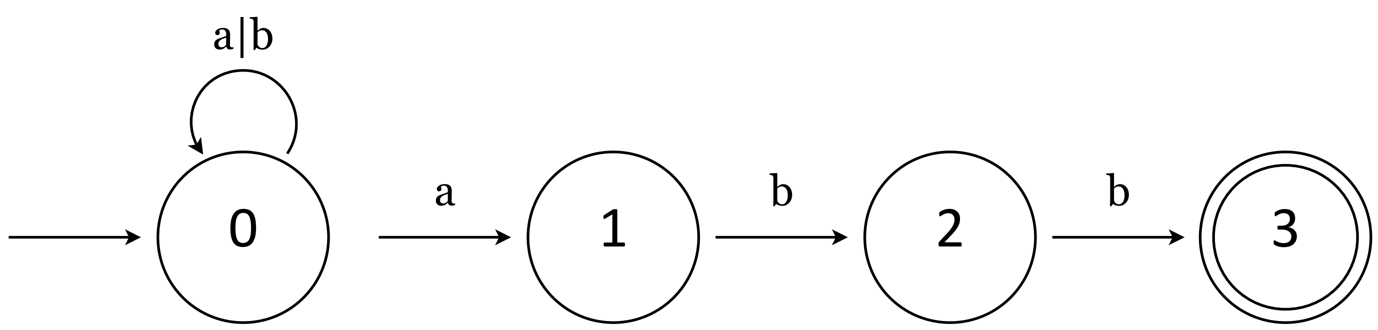


	a	b
{0}	{0,1}	{0}
{0,1}	{0,1}	{0,2}
{0,2}	{0,1}	{0,3}
{0,3}	{0,1}	{0}

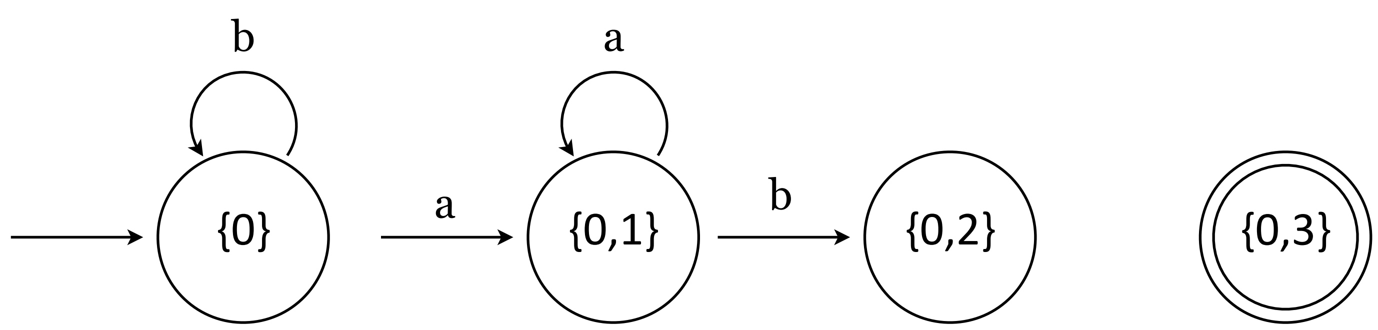


DFA

NFA to DFA via Subset Construction

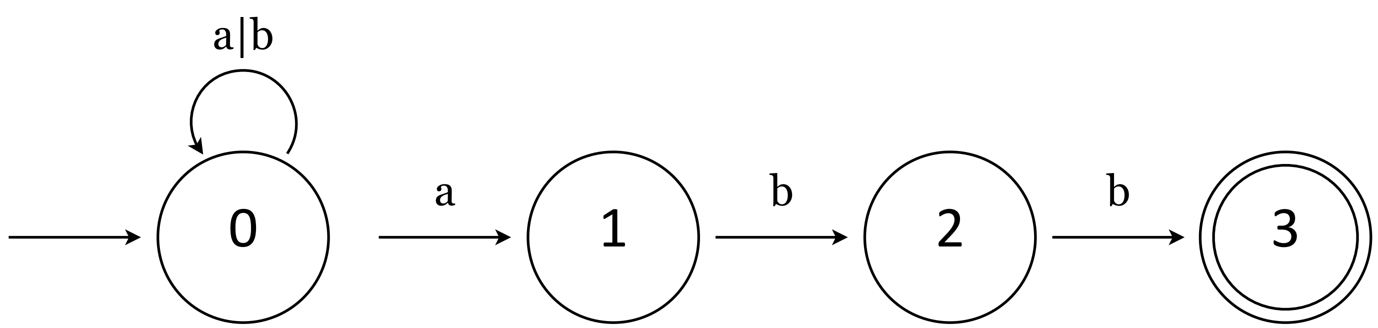


	a	b
{0}	{0,1}	{0}
{0,1}	{0,1}	{0,2}
{0,2}	{0,1}	{0,3}
{0,3}	{0,1}	{0}

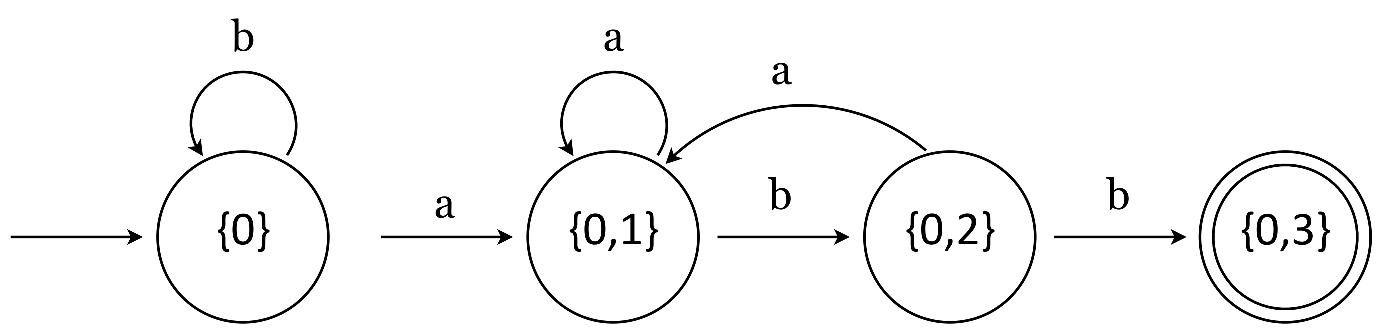


DFA

NFA to DFA via Subset Construction

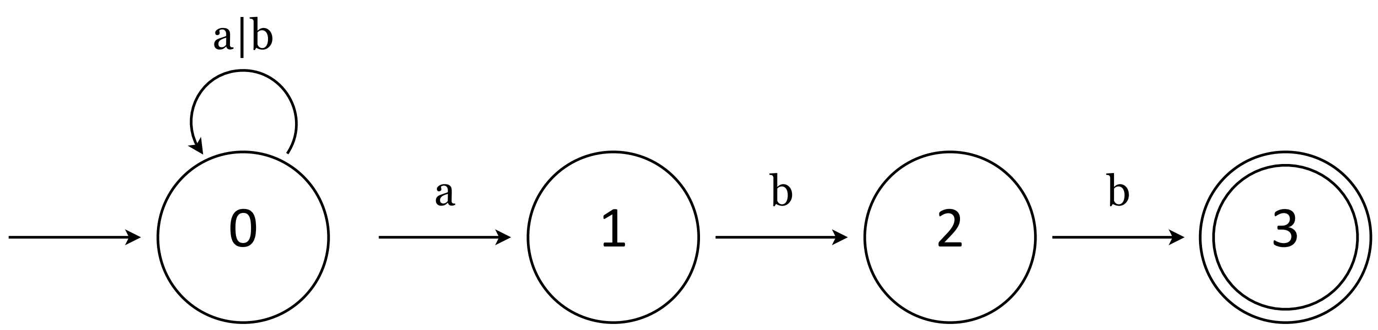


	a	b
{0}	{0,1}	{0}
{0,1}	{0,1}	{0,2}
{0,2}	{0,1}	{0,3}
{0,3}	{0,1}	{0}

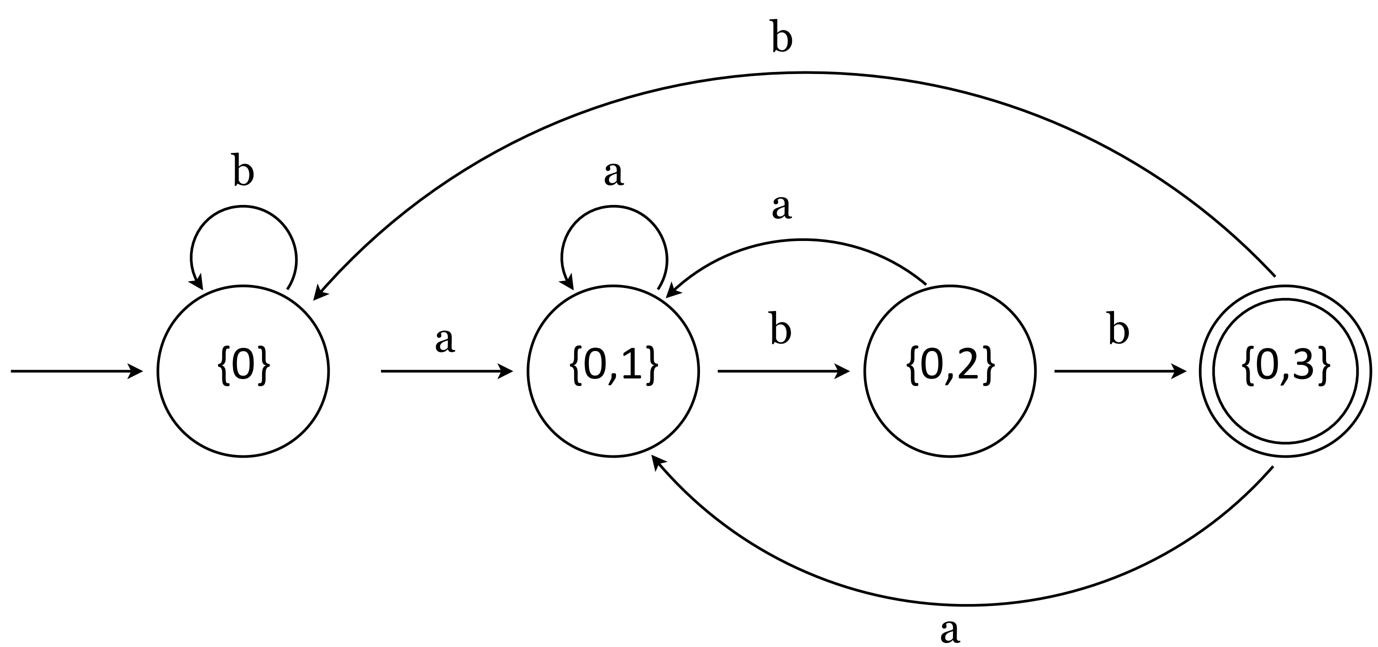


DFA

NFA to DFA via Subset Construction

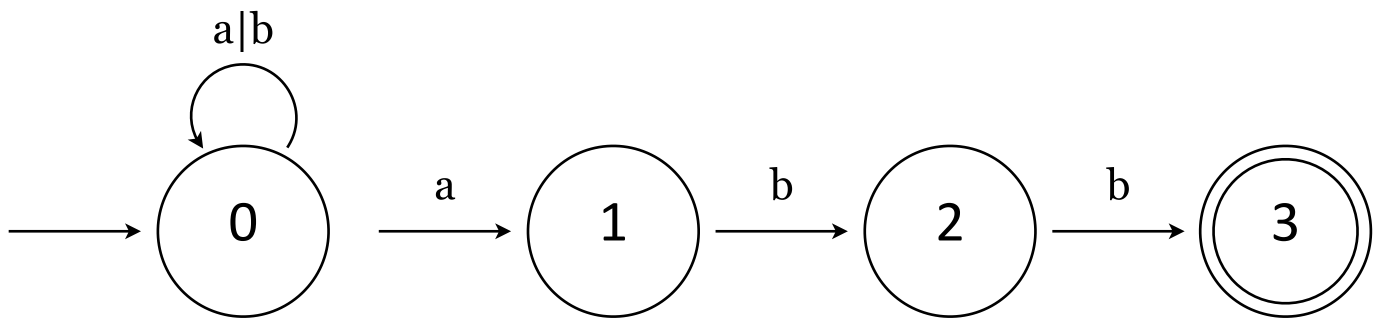


	a	b
{0}	{0,1}	{0}
{0,1}	{0,1}	{0,2}
{0,2}	{0,1}	{0,3}
{0,3}	{0,1}	{0}

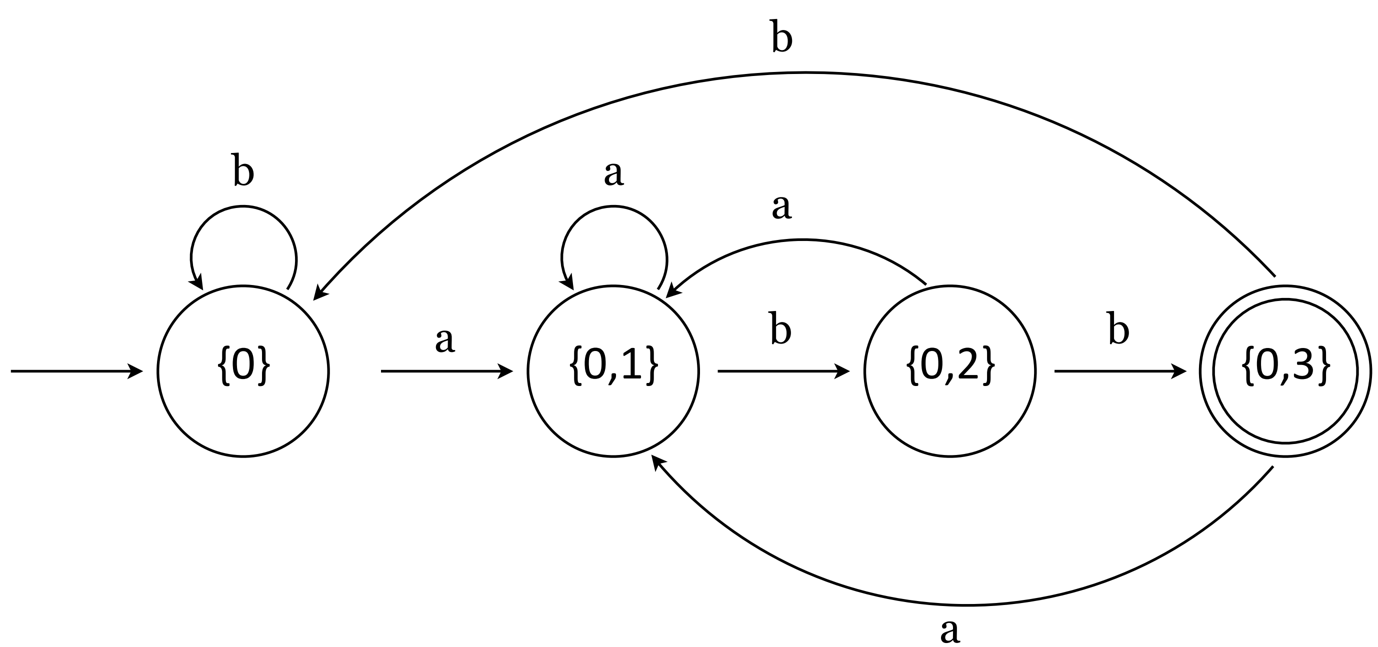


DFA

NFA to DFA via Subset Construction



	a	**b**
{0}	{0,1}	{0}
{0,1}	{0,1}	{0,2}
{0,2}	{0,1}	{0,3}
{0,3}	{0,1}	{0}



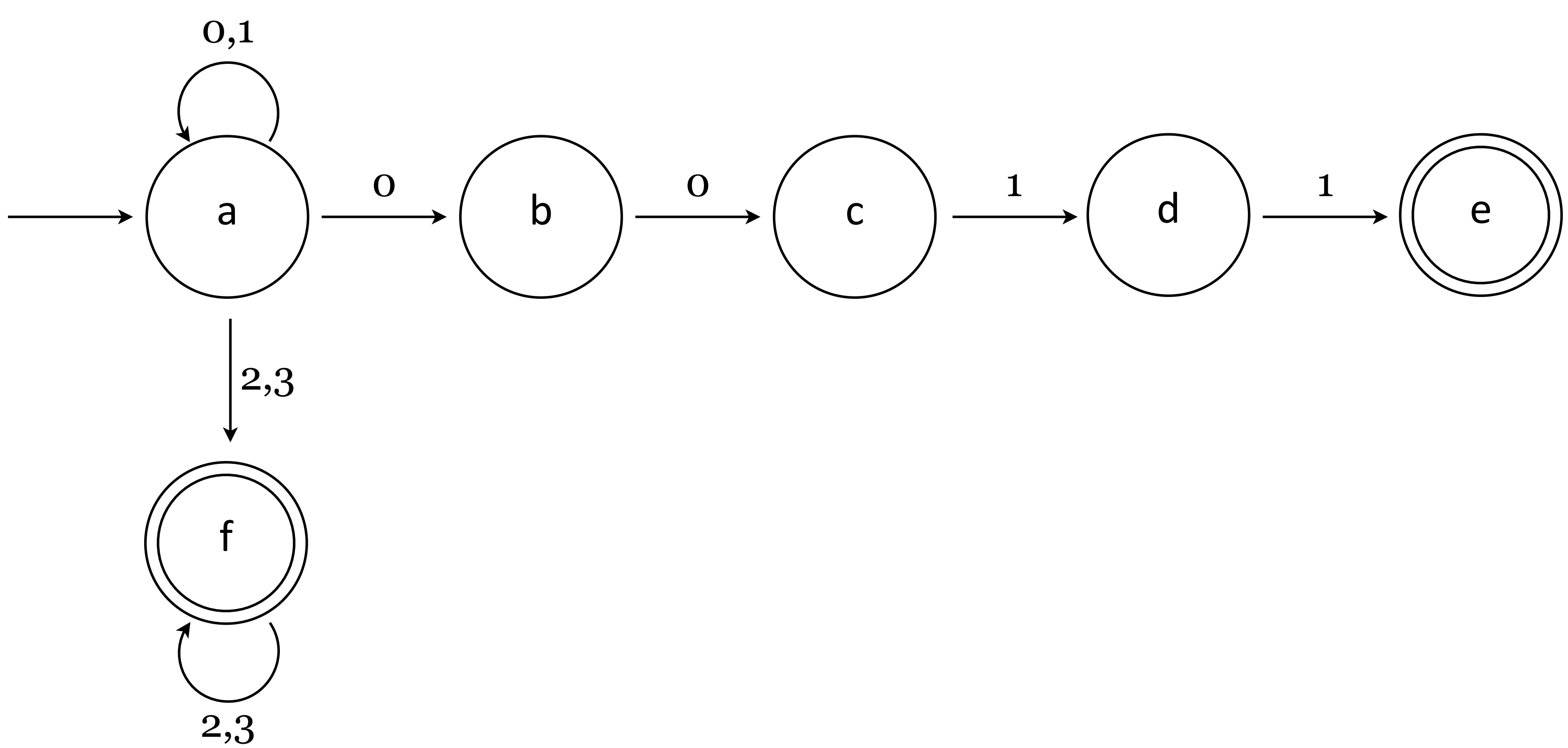
DFA

No transitions on an empty string.
For each state *s* and input symbol *a*, there is only one edge labeled *a* leaving *s*.
I.e., all transitions from one state to another are unique and unambiguous.

Another Example

RegEx $((0|1)^* (2|3)^+) \mid 0\,0\,1\,1$

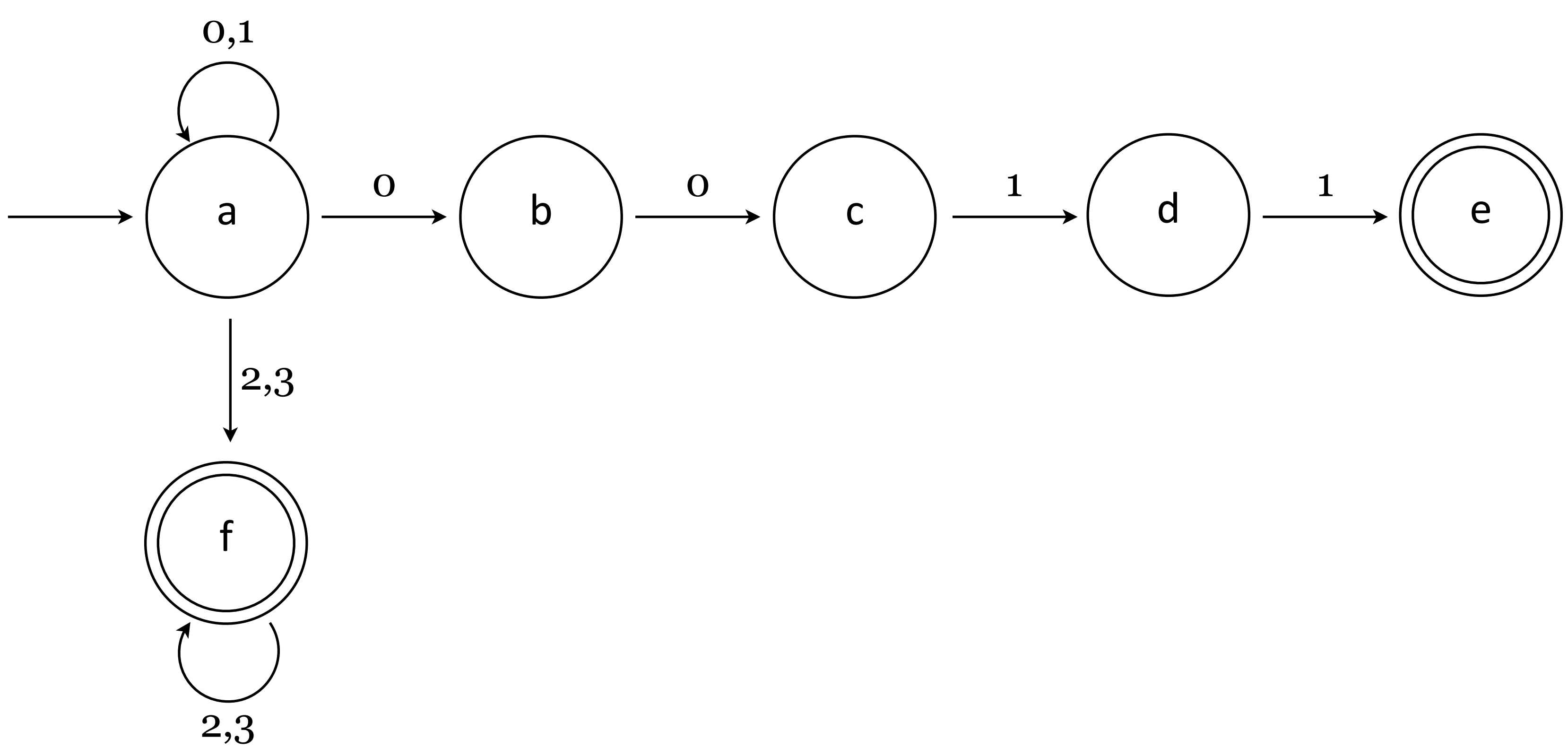
NFA



Another Example

RegEx $\quad ((0|1)^* (2|3)^+) \mid 0\ 0\ 1\ 1$

NFA

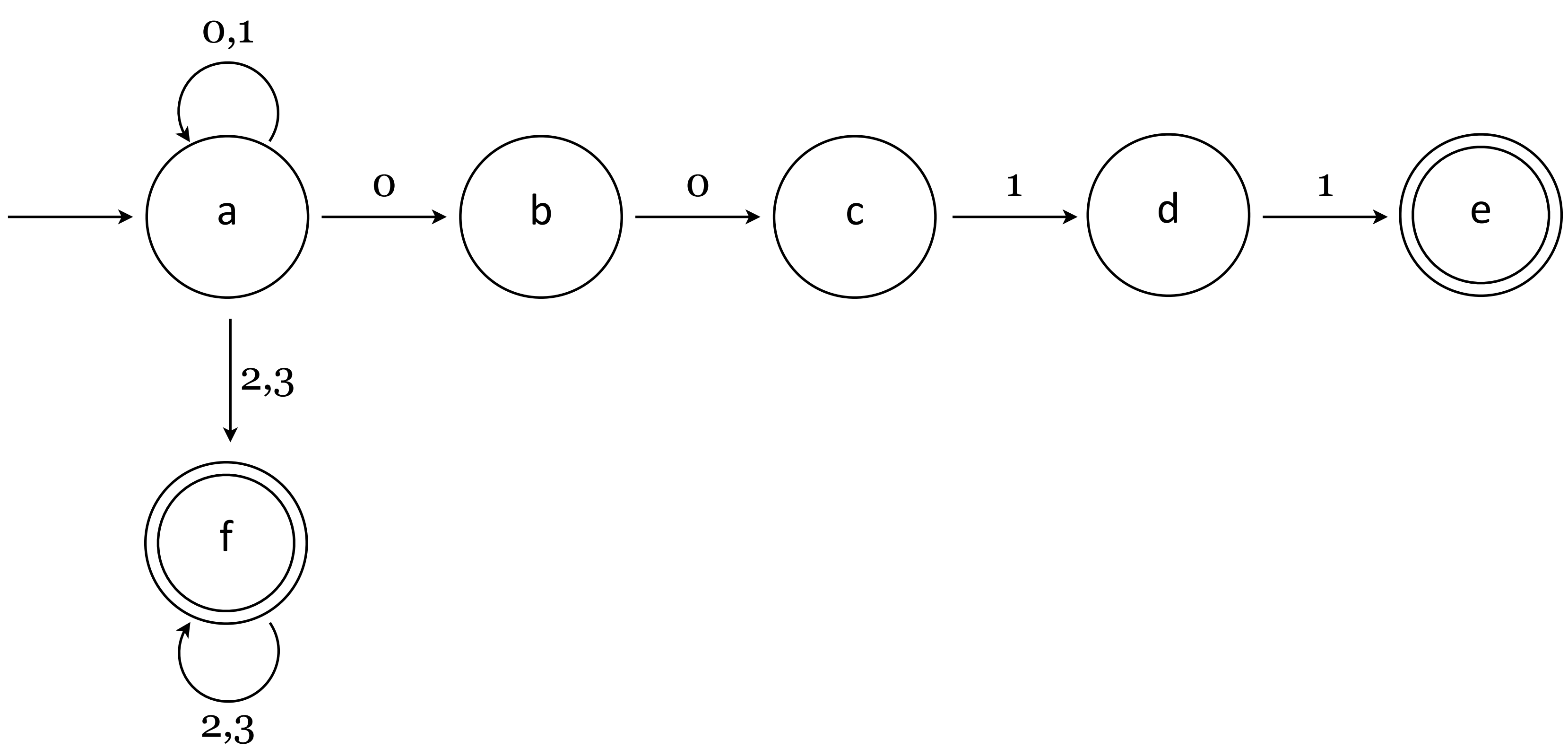


Wait… is that right?

Another Example

RegEx $((0|1)^*(2|3)^+) \mid 0011$

NFA

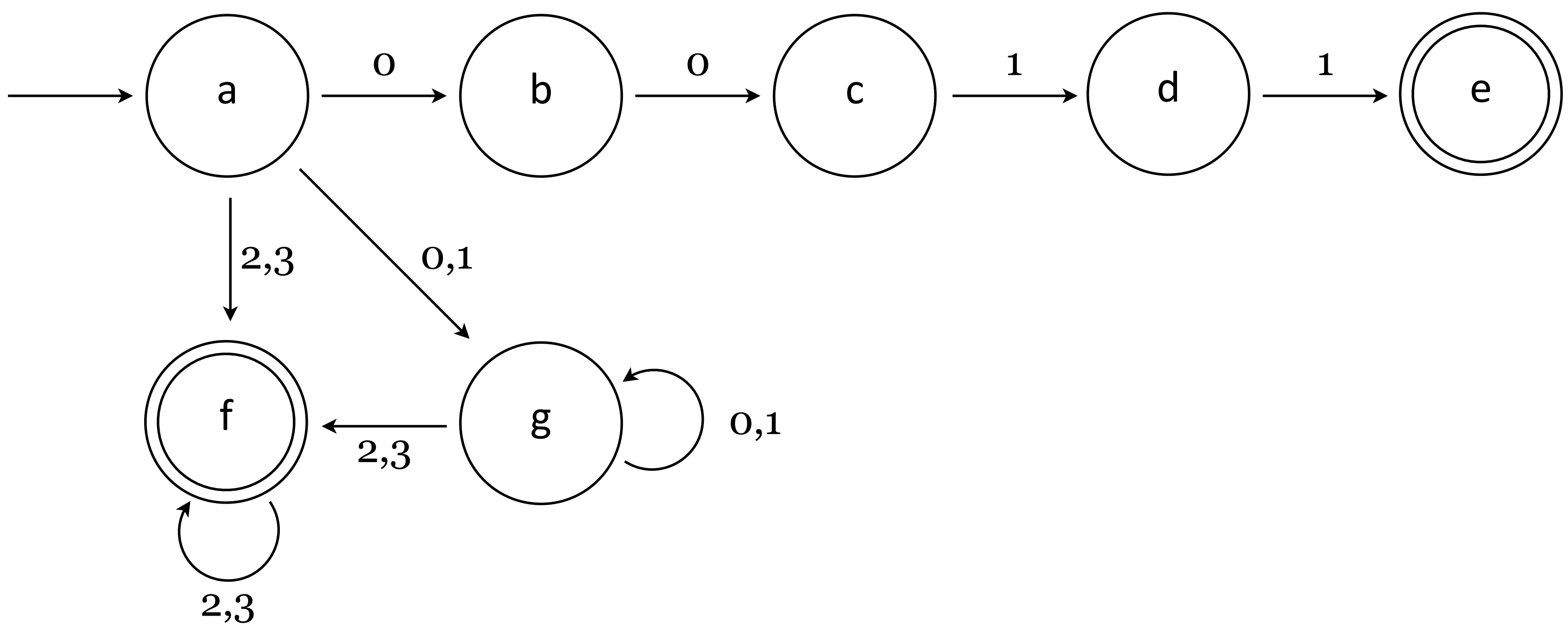


Wait… is that right?
Give me a counter-example.

Another Example

RegEx $(\, (0|1)^* \, (2|3)^+ \,) \;|\; 0\,0\,1\,1$

NFA



Better?