Syntax and Parsing

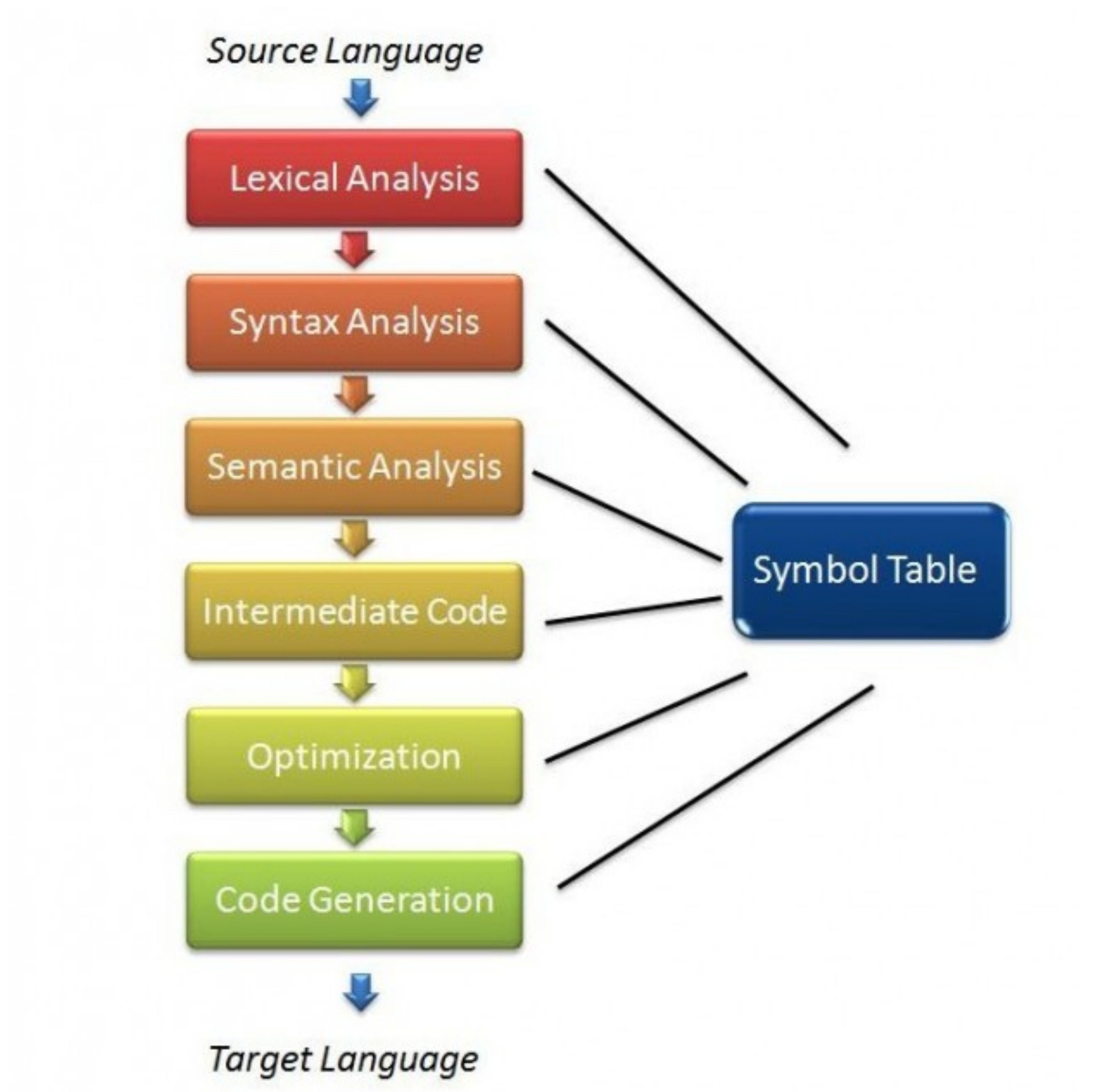
Part 1

At this point in the course, we're going to start to learn how PLs work under the hood

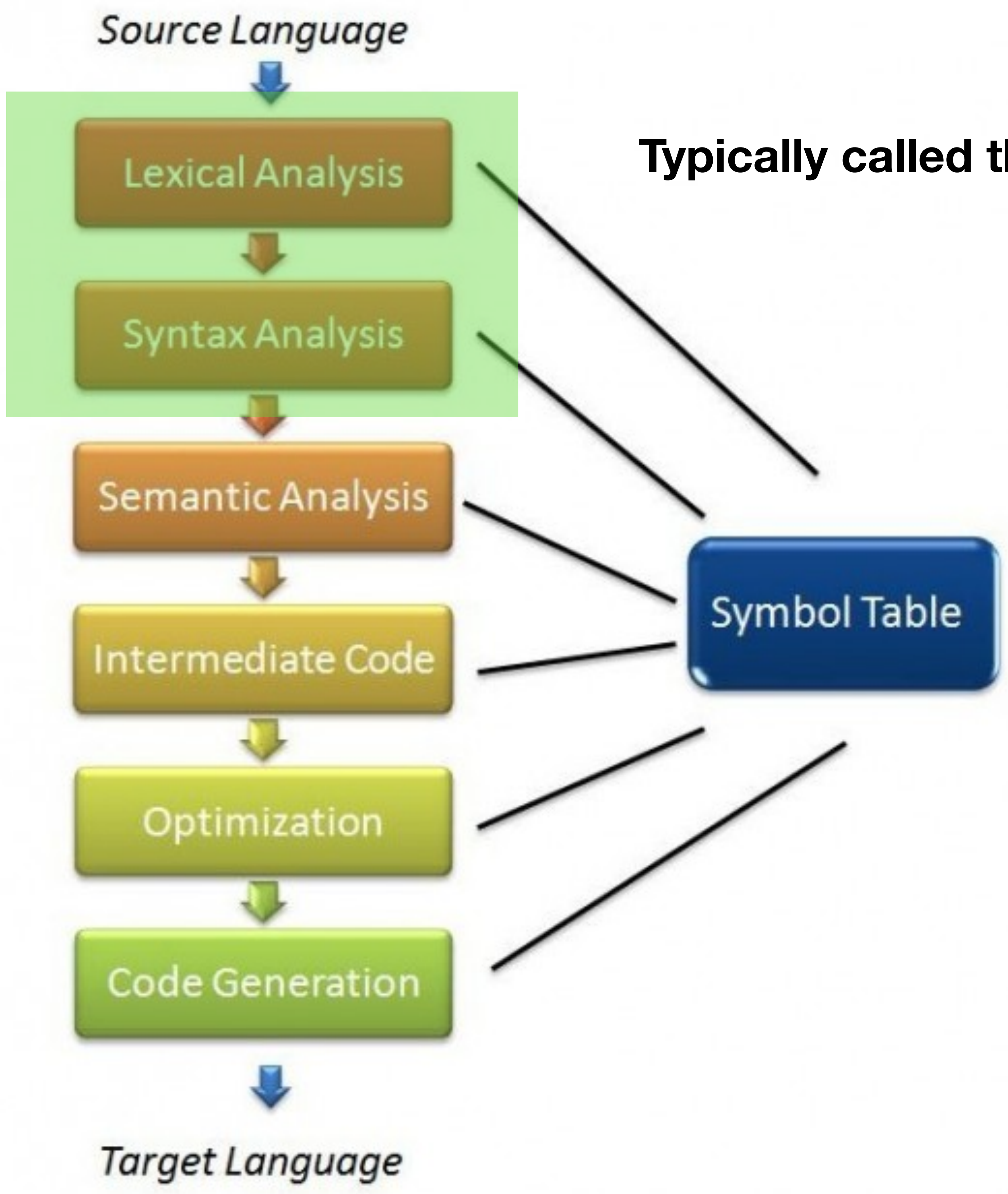
Programming languages take us from raw text on the screen to bits flipping on the processor

Languages are implemented in **phases**

The raw text on the screen is gradually converted to a language the computer speaks http://durofy.com/phases-of-compiler-design/



http://durofy.com/phases-of-compiler-design/



Typically called the front end

The job of the compiler / interpreter’s **front end** is to break down the raw text into a structure that is easier to work with programmatically

This results in an **intermediate representation**

The job of the compiler / interpreter's **front end** is to break down the raw text into a structure that is easier to work with programmatically

This results in an **intermediate representation**

Why?

The job of the compiler / interpreter’s **front end** is to break down the raw text into a structure that is easier to work with programmatically

This results in an **intermediate representation**

Why?

Working on raw text way too kludgey!

Don’t get too hung up on specifics right now, we’ll be **implementing** one programming language (Forth) soon!

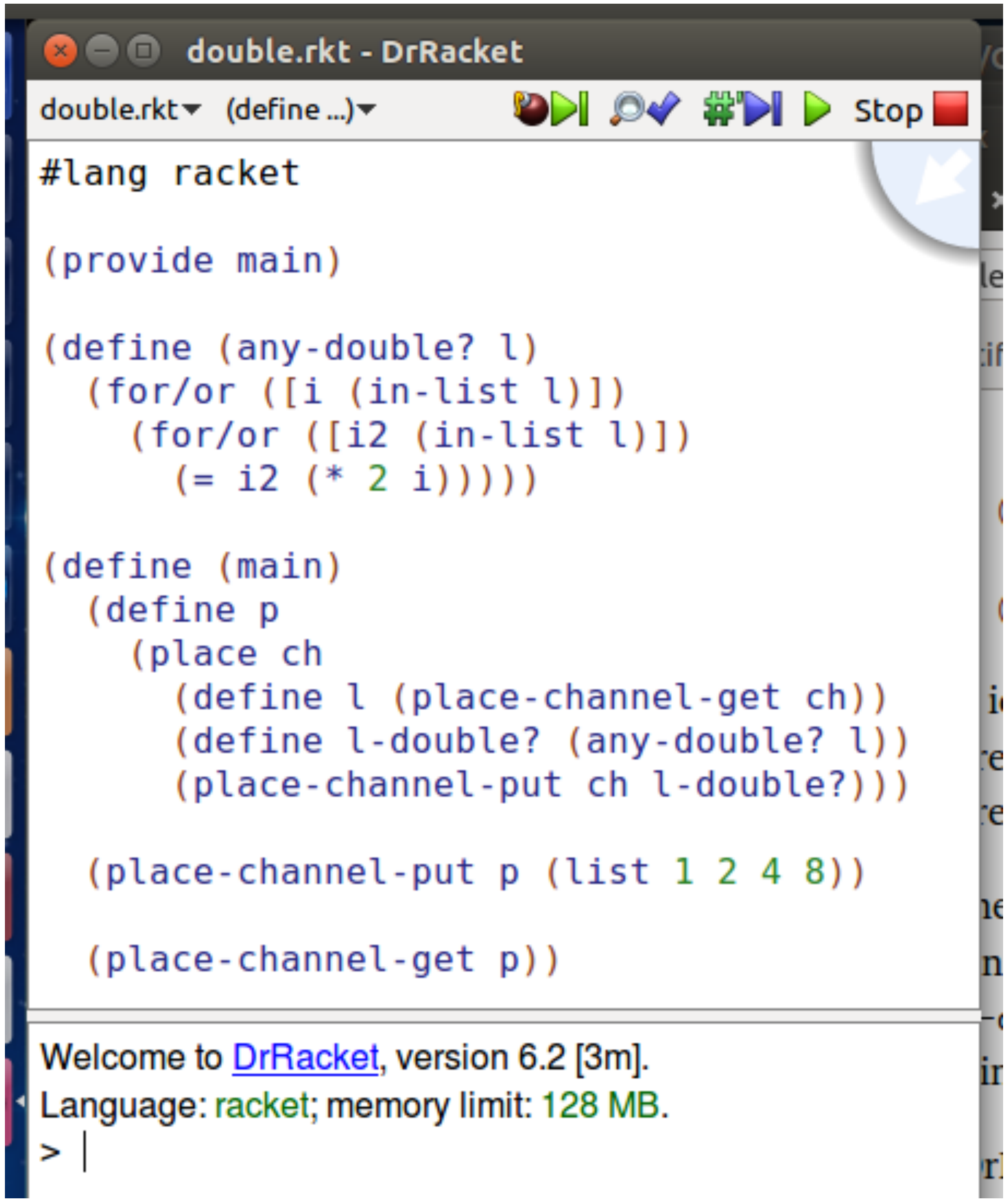
Today we’re going to focus on **lexical analysis**

I.e., how do we break up raw text into a stream of tokens?

Or, how do I define a token?

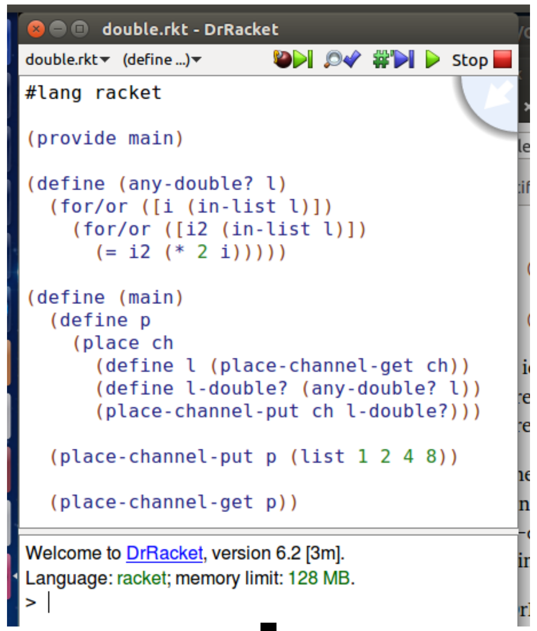
Next lecture we'll talk about combining these raw tokens to build up a **grammar**

This will help us define the syntax of a PL *compositionally*



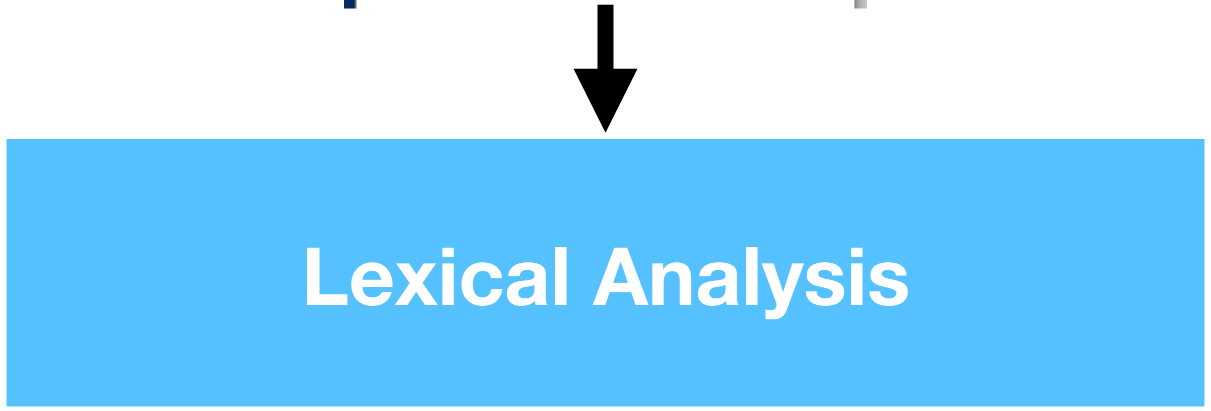
```
#lang racket

(provide main)

(define (any-double? l)
  (for/or ([i (in-list l)])
    (for/or ([i2 (in-list l)])
      (= i2 (* 2 i)))))

(define (main)
  (define p
    (place ch
      (define l (place-channel-get ch))
      (define l-double? (any-double? l))
      (place-channel-put ch l-double?)))

  (place-channel-put p (list 1 2 4 8))

  (place-channel-get p))
```

```
Welcome to DrRacket, version 6.2 [3m].
Language: racket; memory limit: 128 MB.
> 
```



```
#lang racket

(provide main)

(define (any-double? l)
  (for/or ([i (in-list l)])
    (for/or ([i2 (in-list l)])
      (= i2 (* 2 i)))))

(define (main)
  (define p
    (place ch
      (define l (place-channel-get ch))
      (define l-double? (any-double? l))
      (place-channel-put ch l-double?)))

  (place-channel-put p (list 1 2 4 8))

  (place-channel-get p))
```

```
Welcome to DrRacket, version 6.2 [3m].
Language: racket; memory limit: 128 MB.
> 
```



Lexical analysis breaks apart a (potentially huge) file into sequence of **tokens**

Token: atomic piece of syntax of a language

```
(define (hello-world)
    (display “Hello, world!\n”))
```

$$\Big\downarrow$$

LPAREN ID(“define”) LPAREN Identifier(“hello-world”)
RPAREN LPAREN ID(“display”) STRING(“Hello,
world\n”) RPAREN RPAREN

One example of a token *stream*

```
(define (hello-world)
    (display “Hello, world!\n”))
```

$\Big\downarrow$ Lexical analysis

LPAREN ID(“define”) LPAREN Identifier(“hello-world”) RPAREN LPAREN ID(“display”) STRING(“Hello, world\n”) RPAREN RPAREN

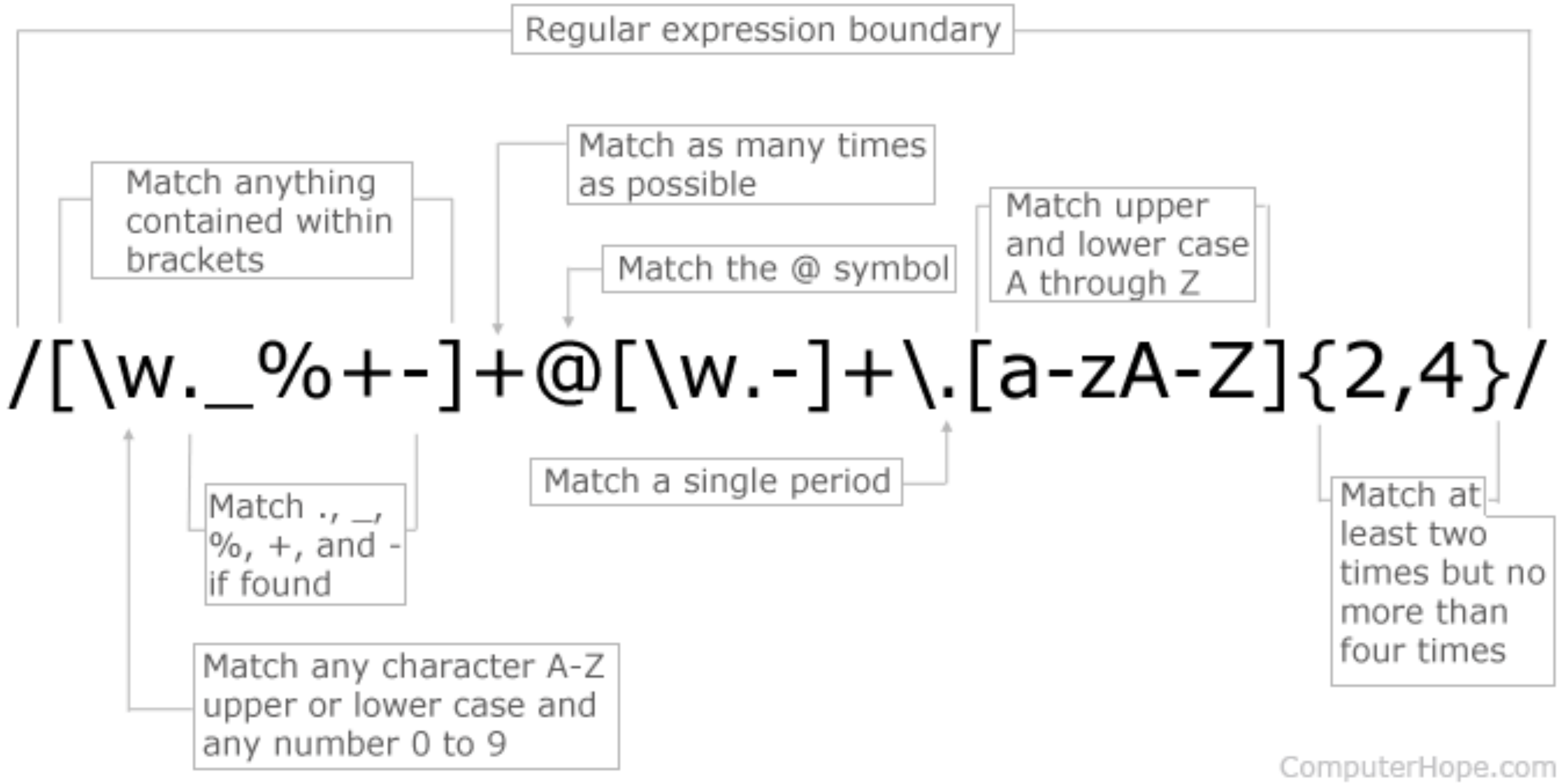
Enter: Regular Expressions

Regular expressions are basically **string matchers**

A regular expression classifies strings into two categories

Accept or **reject**

Regular Expression E-mail Matching Example


Regular expression boundary
Match anything contained within brackets
Match as many times as possible
Match the @ symbol
Match upper and lower case A through Z
/[\w._%+-]+@[\w.-]+\.[a-zA-Z]{2,4}/
Match ., _, %, +, and - if found
Match a single period
Match at least two times but no more than four times
Match any character A-Z upper or lower case and any number 0 to 9

ComputerHope.com

Regular expressions are a **general device** in computing, but there are many **implementations**

They each vary a bit, so read the docs on whatever language you’re using

(Kris now talks about basic building blocks of regexes: constants, concat, Kleene star, union, using () for grouping)

Talk about derived forms: [a-z], {a,b,c}, a+

The “language” of a regex is the set of strings it accepts

What is this language?

```
(0|1)*
```

What about this one?

```
1(0)*
```

How about this one

```
((0|1)(0|1)(0|1))*
```

Write “the set of odd binary strings” as a regex

Write “an odd number of **b**s followed by an even number of **a**s”

“Any number of 1s, followed by an even number of 0s, followed by a single 1”

Regular expressions classify the so called **regular languages**