AN INTRODUCTION TO

STACK BASED LANGUAGES

USING THE

FACTOR PROGRAMMING LANGUAGE

ABOUT FACTOR

ABOUT FACTOR

- Relatively New (started in 2003)
- Multi-paradigm: Stack Based, Functional, OOP
- Interactive: Integrated IDE + REPL
- Thin Line between Language & Library

ABOUT FACTOR

- Relatively New (started in 2003)
- Multi-paradigm: Stack Based, Functional, OOP
- Interactive: Integrated IDE + REPL
- Thin Line between Language & Library

THE BASICS

THE BASICS

THE BASICS

- Programs = Literals + Words

- Literals = Values

```
          "Strings"
123.45              objects
          { 1 2 3 }
```

- Words are named blocks of code

THE BASICS (WORDS)

THE BASICS (WORDS)

- Words = Functions
- Operate on a Data Stack
- Words have “Stack Effects”
- Stack Effects = Function Signatures

THE BASICS (STACK EFFECTS)

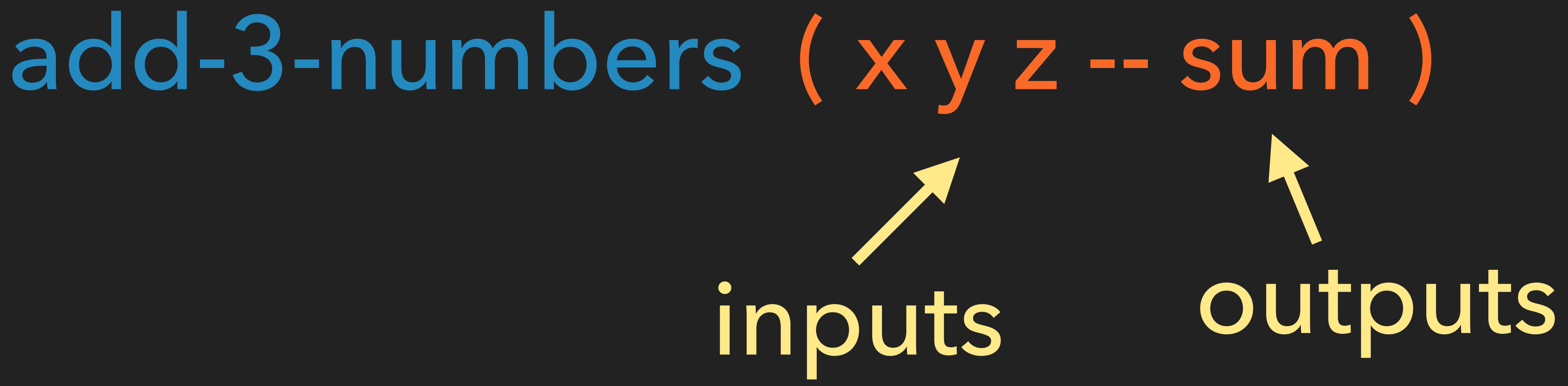
```
add-3-numbers ( x y z -- sum )
```

THE BASICS (STACK EFFECTS)

add-3-numbers (x y z -- sum)

inputs

THE BASICS (STACK EFFECTS)



```
add-3-numbers ( x y z -- sum )
```

inputs

outputs

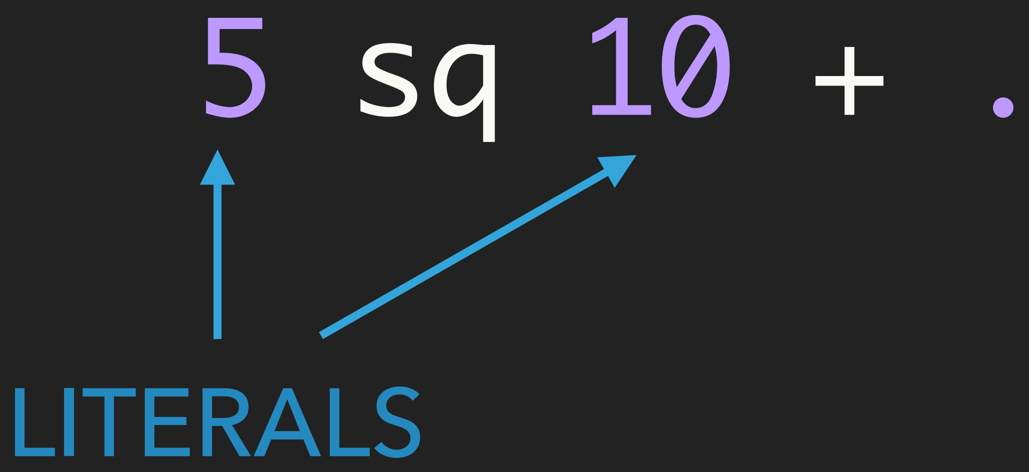
THE BASICS

Example:

```
5 sq 10 + .
```

THE BASICS

Example:

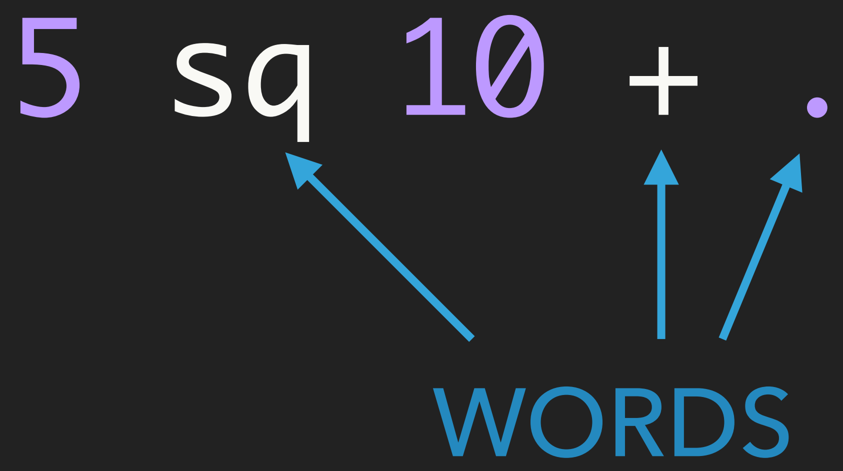


```
5 sq 10 + .
```

LITERALS

THE BASICS

Example:

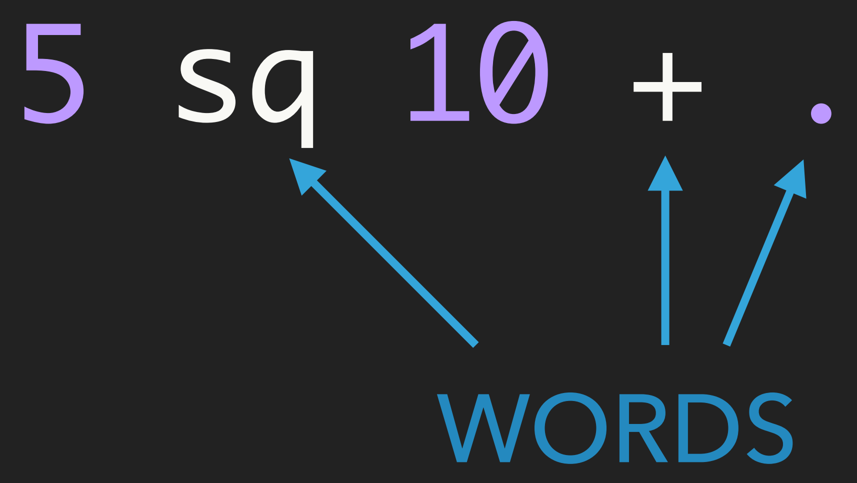


```
5 sq 10 + .
```

WORDS

THE BASICS

Example:



```
5 sq 10 + .
```

All Valid Word Names

*	even?	<person>	>>>
✓	✓	✓	✓

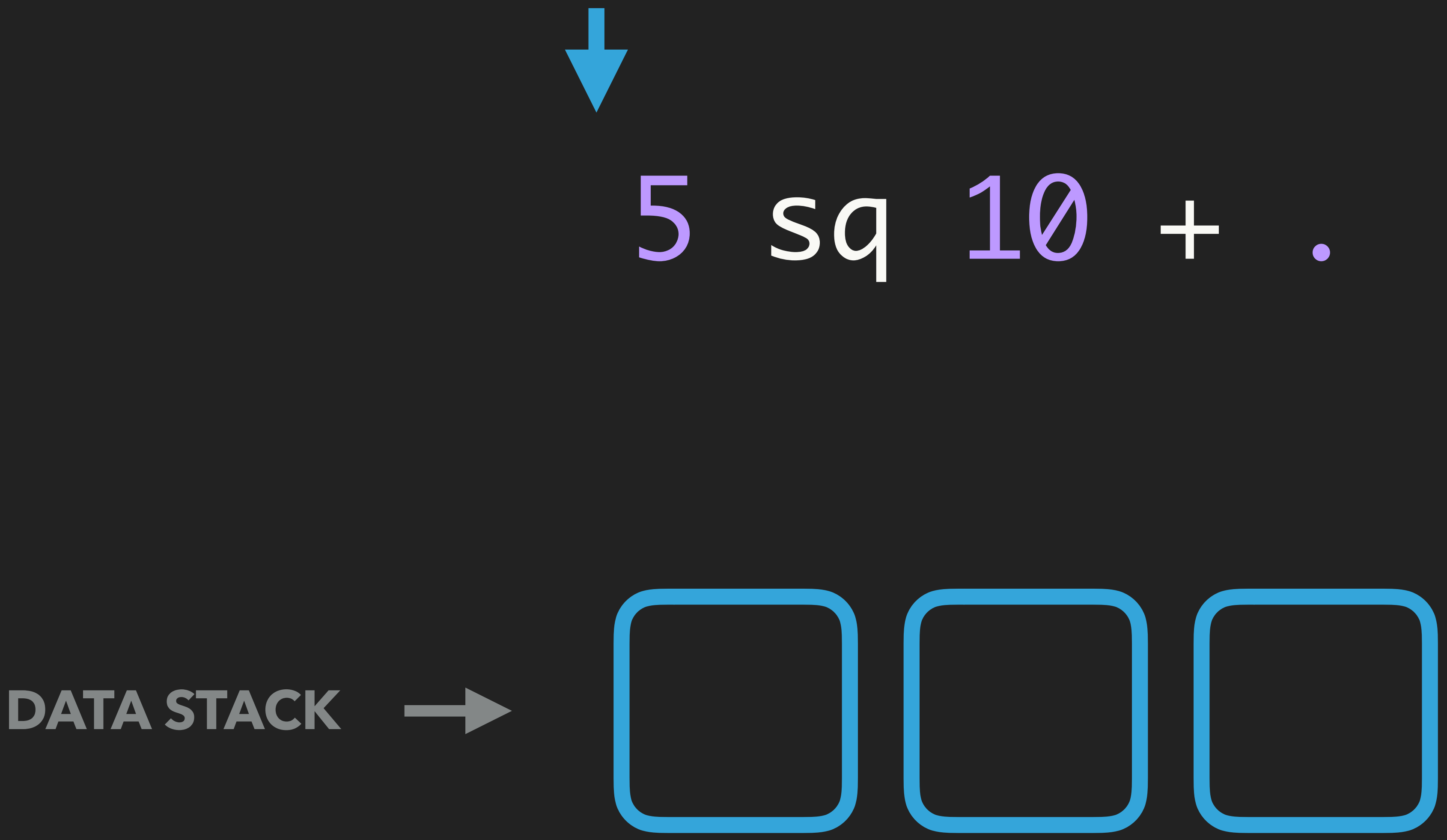
THE BASICS

Example:

```
5 sq 10 + .
```

THE BASICS

Example:

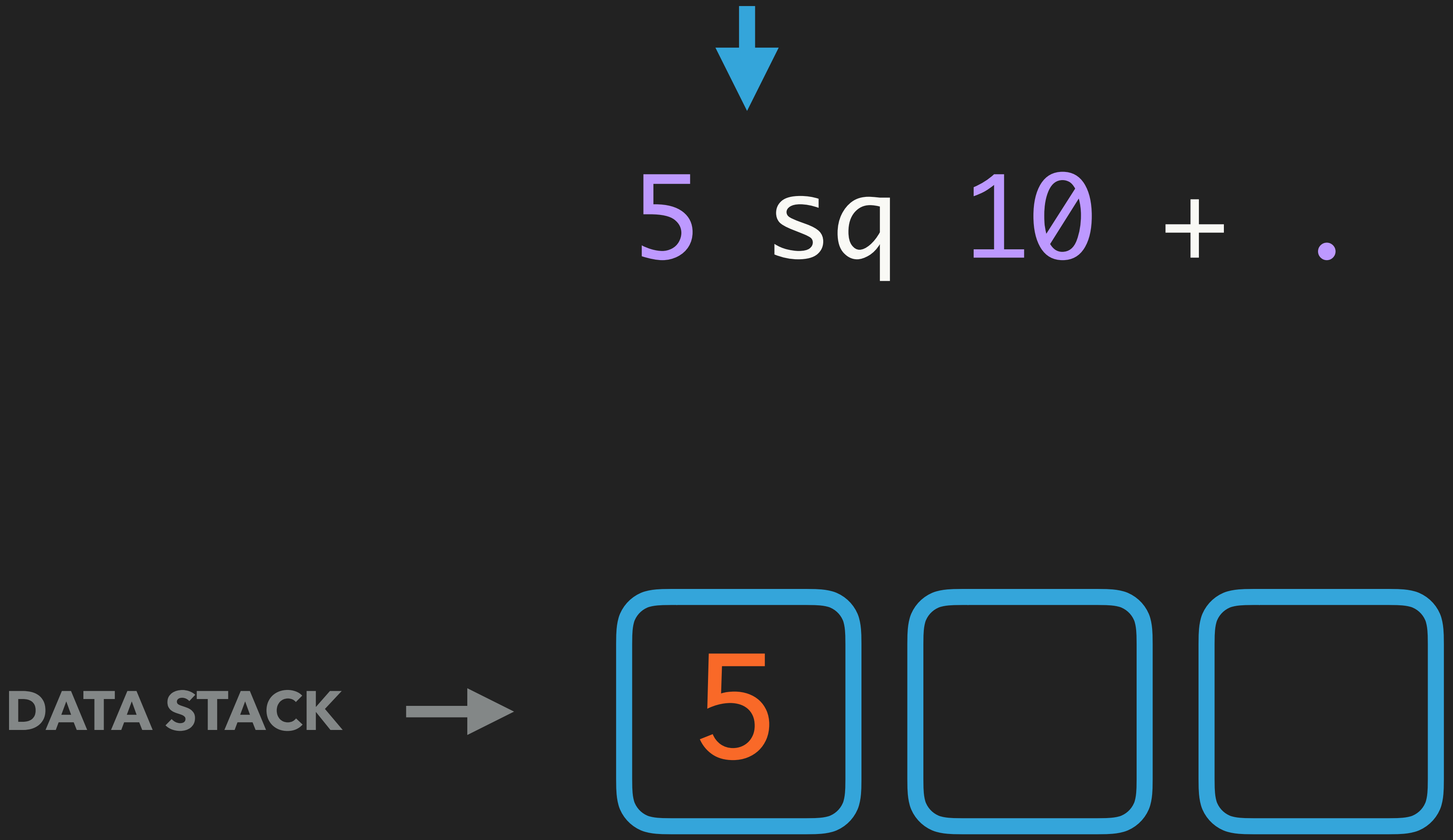


```
5 sq 10 + .
```

DATA STACK

THE BASICS

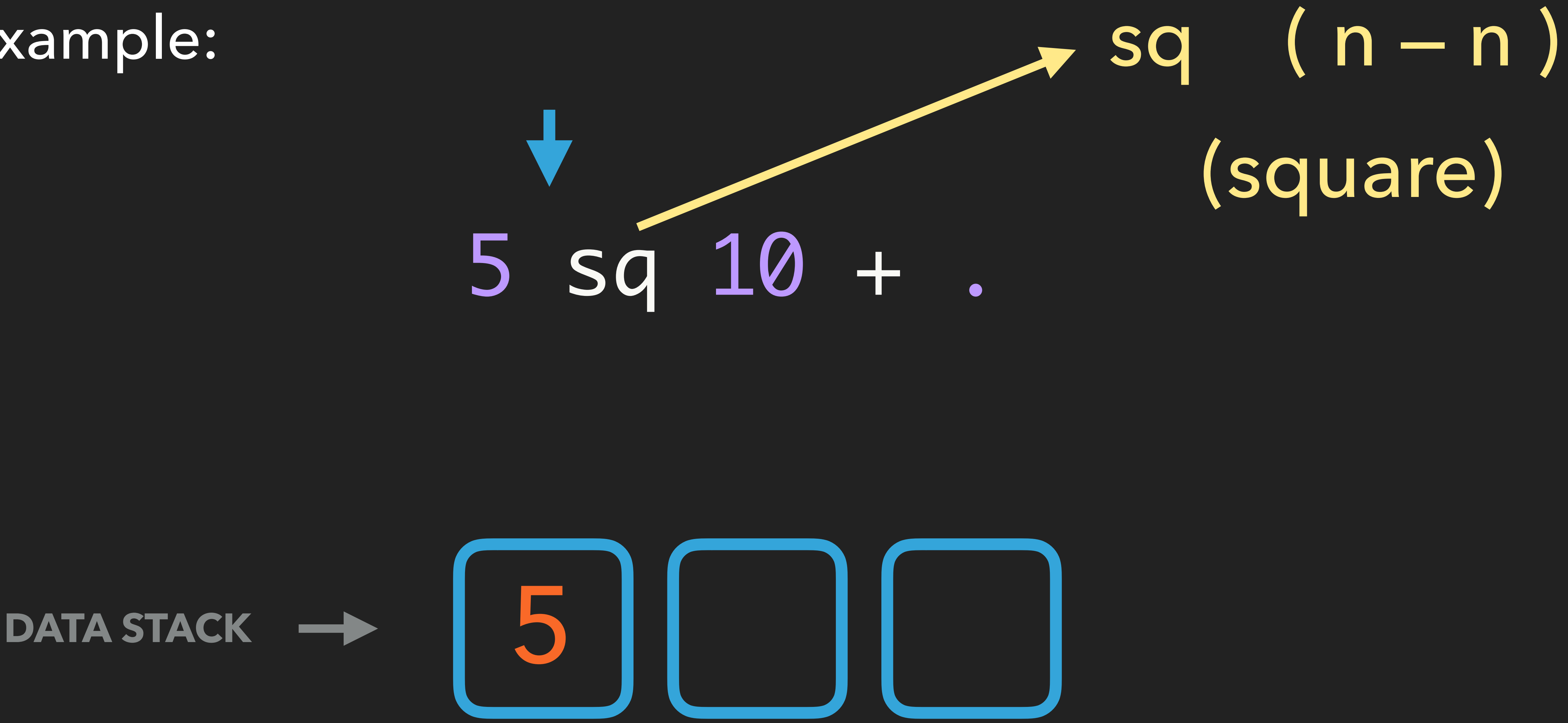
Example:



```
5 sq 10 + .
```

THE BASICS

Example:



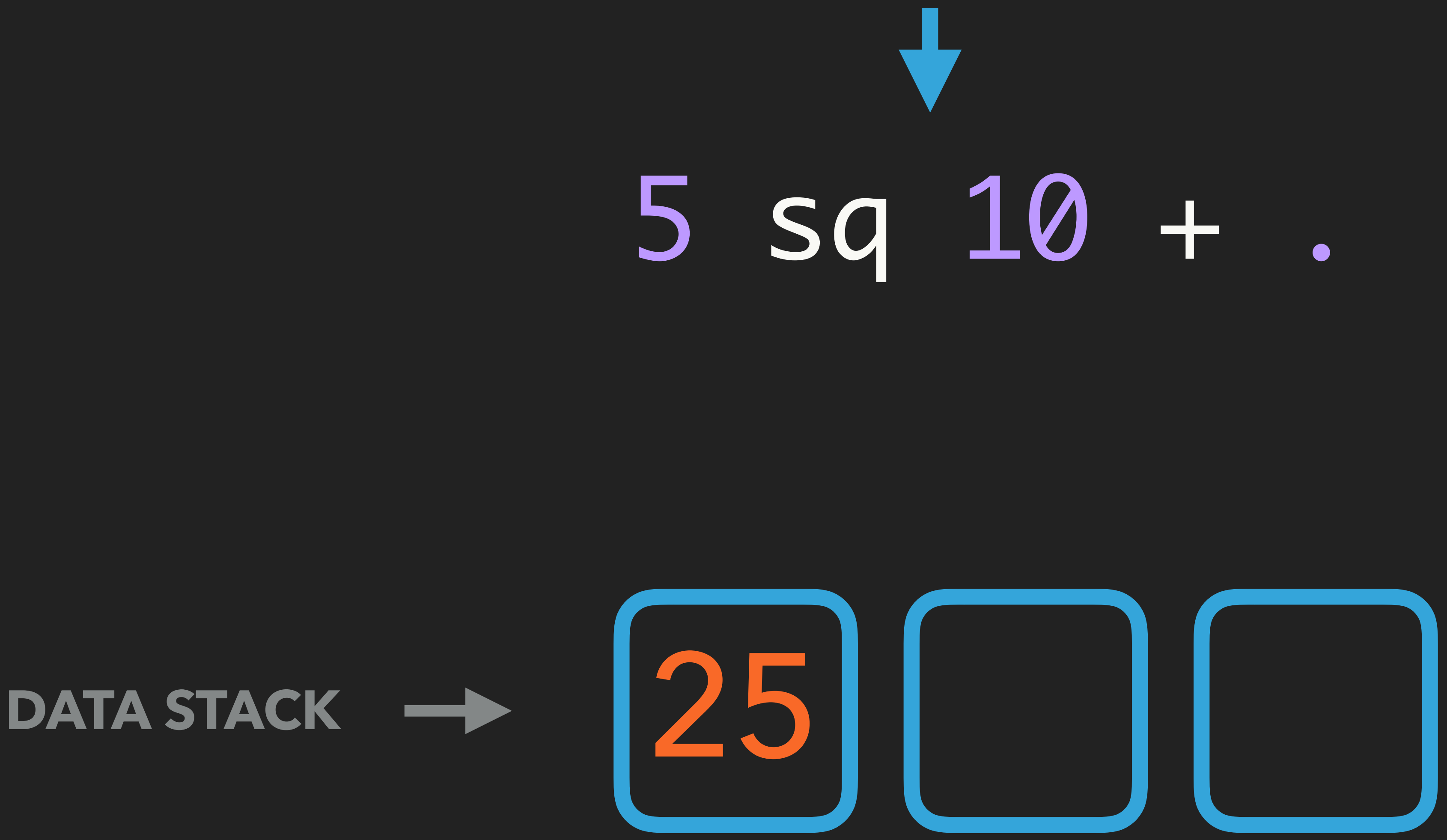
sq (n – n)

(square)

```
5 sq 10 + .
```

THE BASICS

Example:



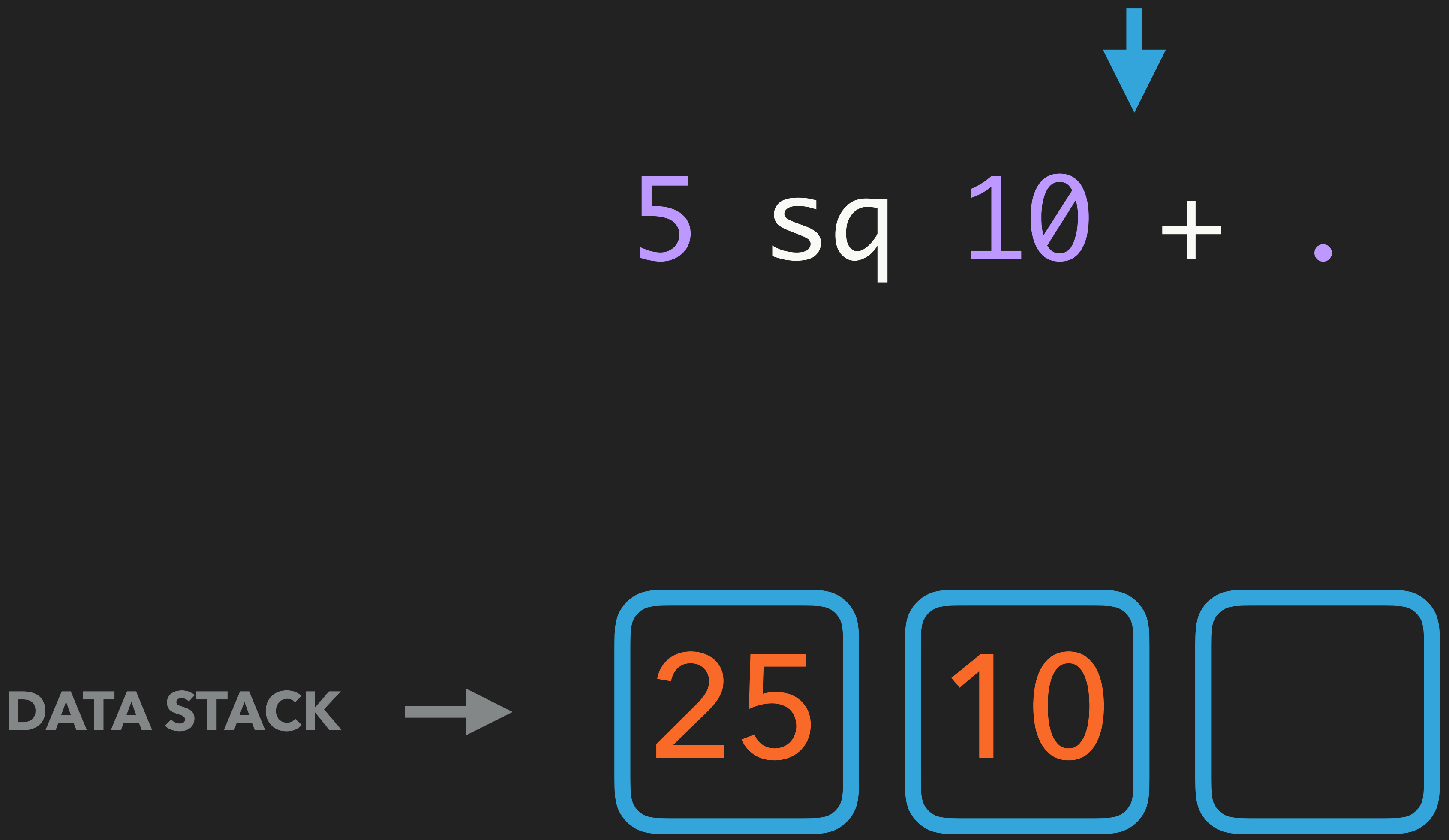
```
5 sq 10 + .
```

DATA STACK

25

THE BASICS

Example:



```
5 sq 10 + .
```

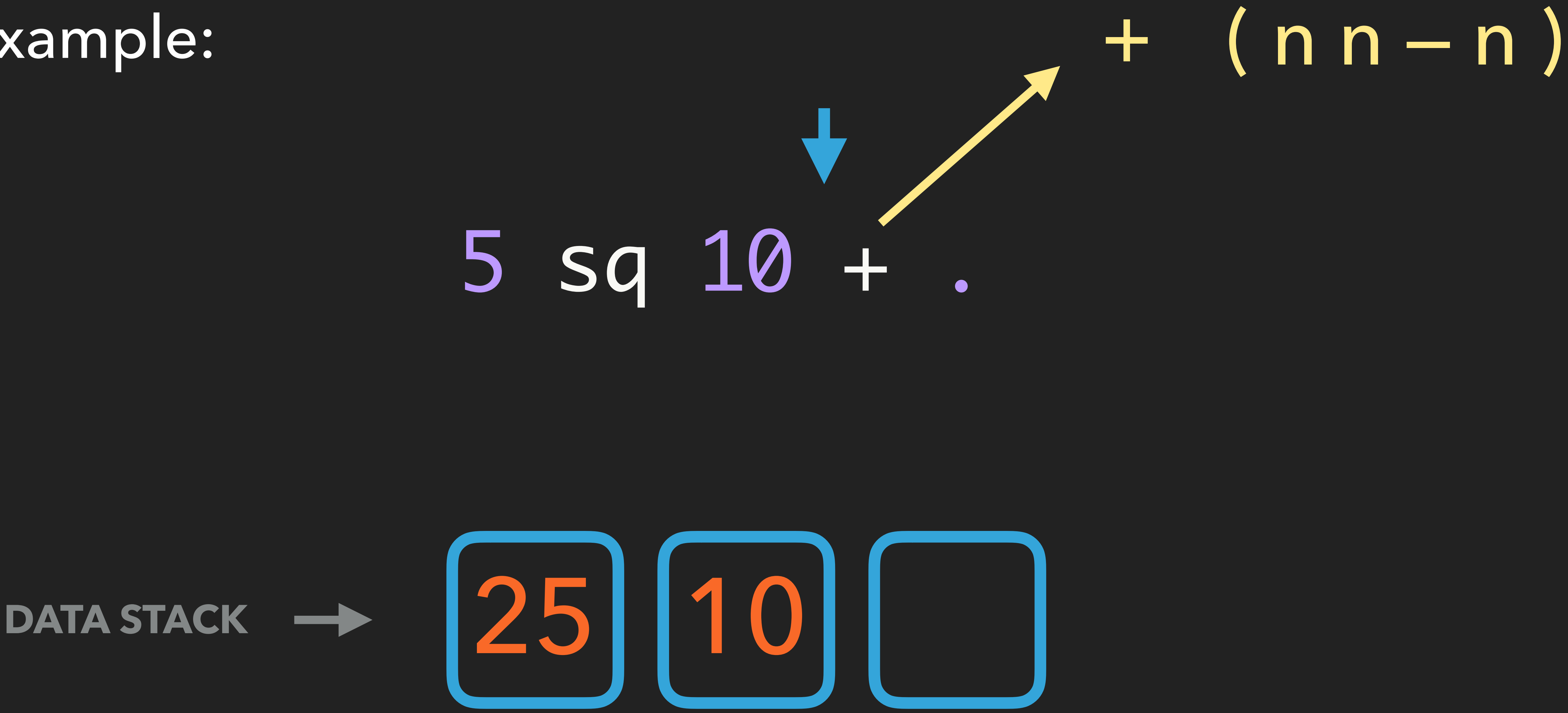
DATA STACK

25

10

THE BASICS

Example:



+ (n n – n)

```
5 sq 10 + .
```

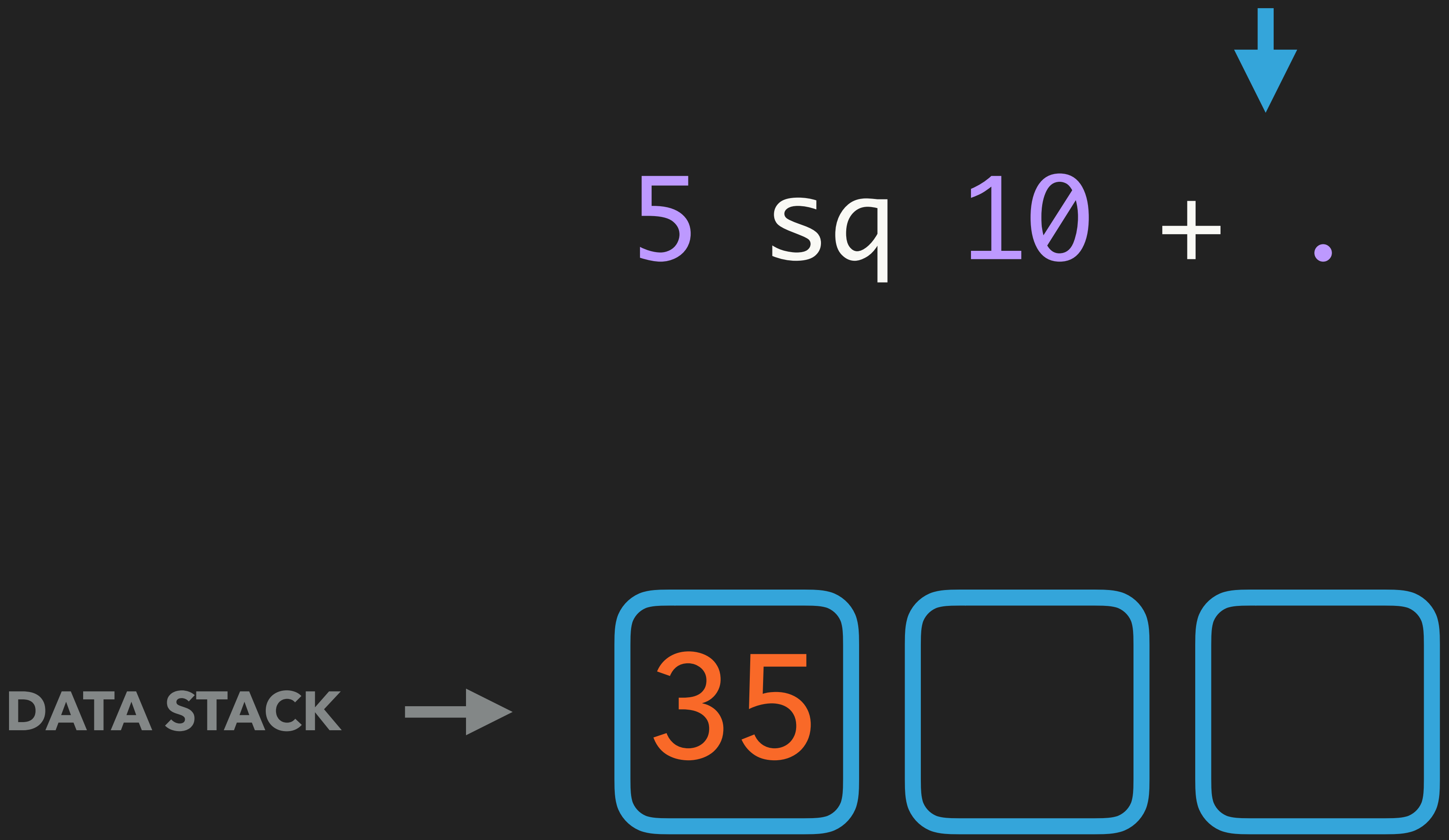
DATA STACK

25

10

THE BASICS

Example:



```
5 sq 10 + .
```

DATA STACK

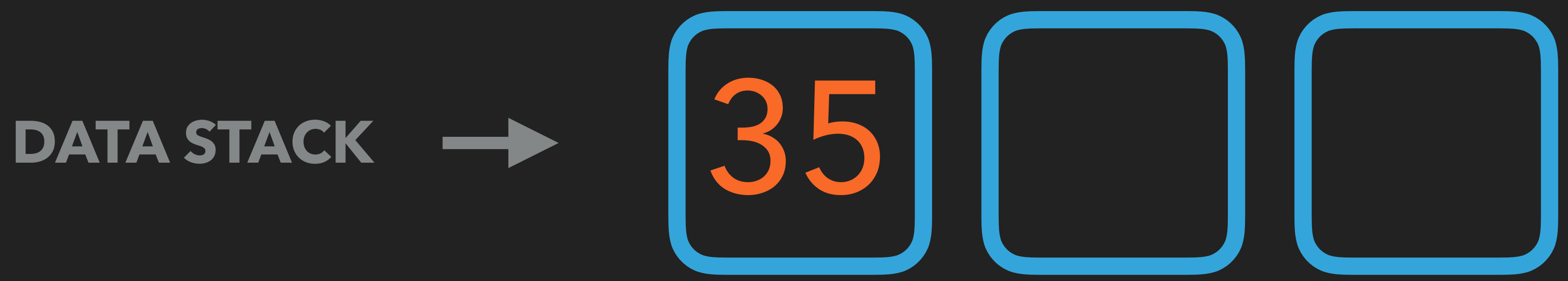
35

THE BASICS

Example:

. (obj –)

```
5 sq 10 + .
```



DATA STACK

THE BASICS

Example:

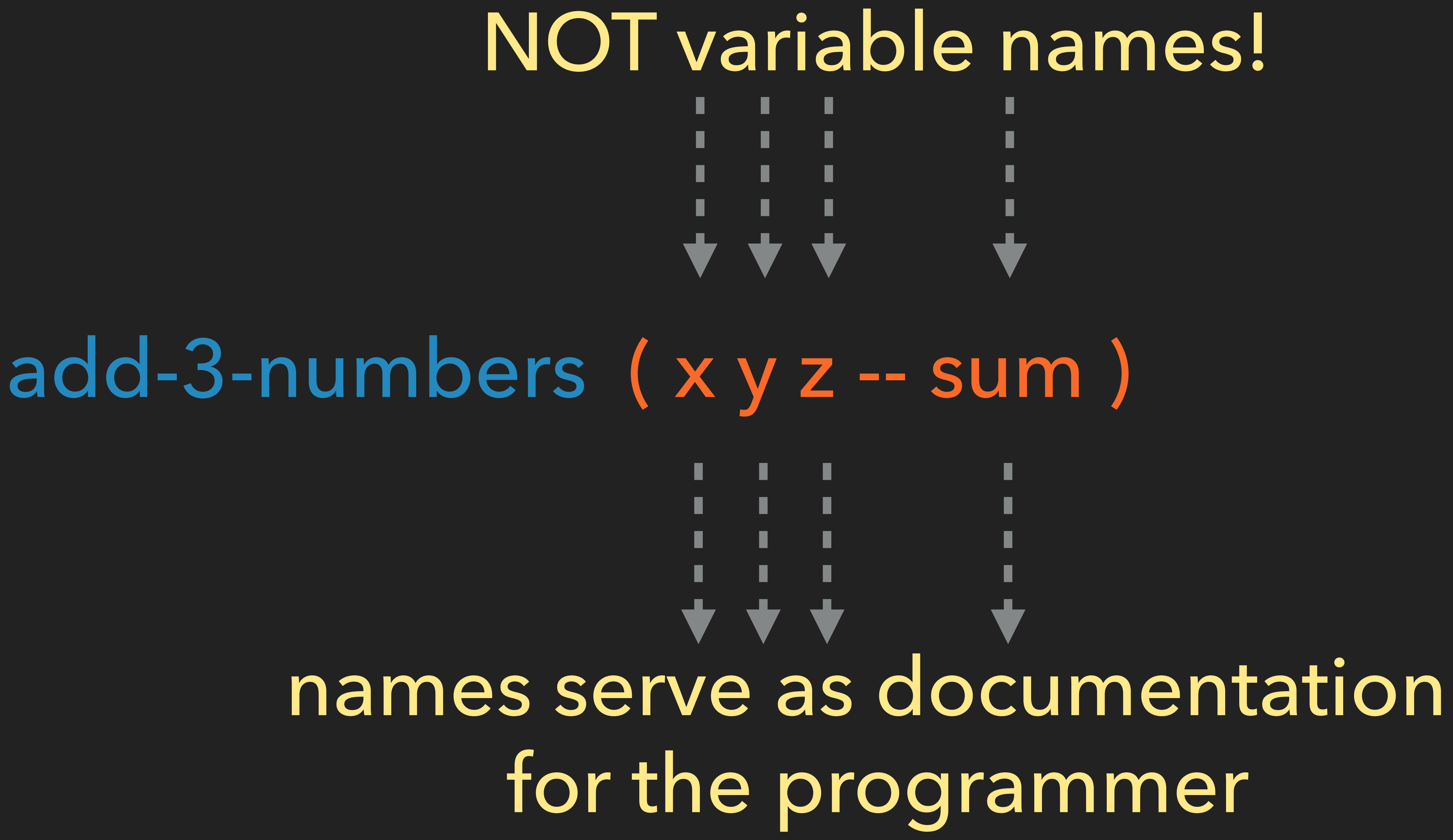
```
5 sq 10 + .
```

(35 is output on console)

THE BASICS (STACK EFFECTS)

```
add-3-numbers  ( x y z -- sum )
```

THE BASICS (STACK EFFECTS)



NOT variable names!

```
add-3-numbers ( x y z -- sum )
```

names serve as documentation
for the programmer

THE BASICS (STACK EFFECTS)

```
add-3-numbers ( x y z -- sum )
```

is the same as

```
add-3-numbers ( x x x -- x )
```

DEFINING NEW WORDS

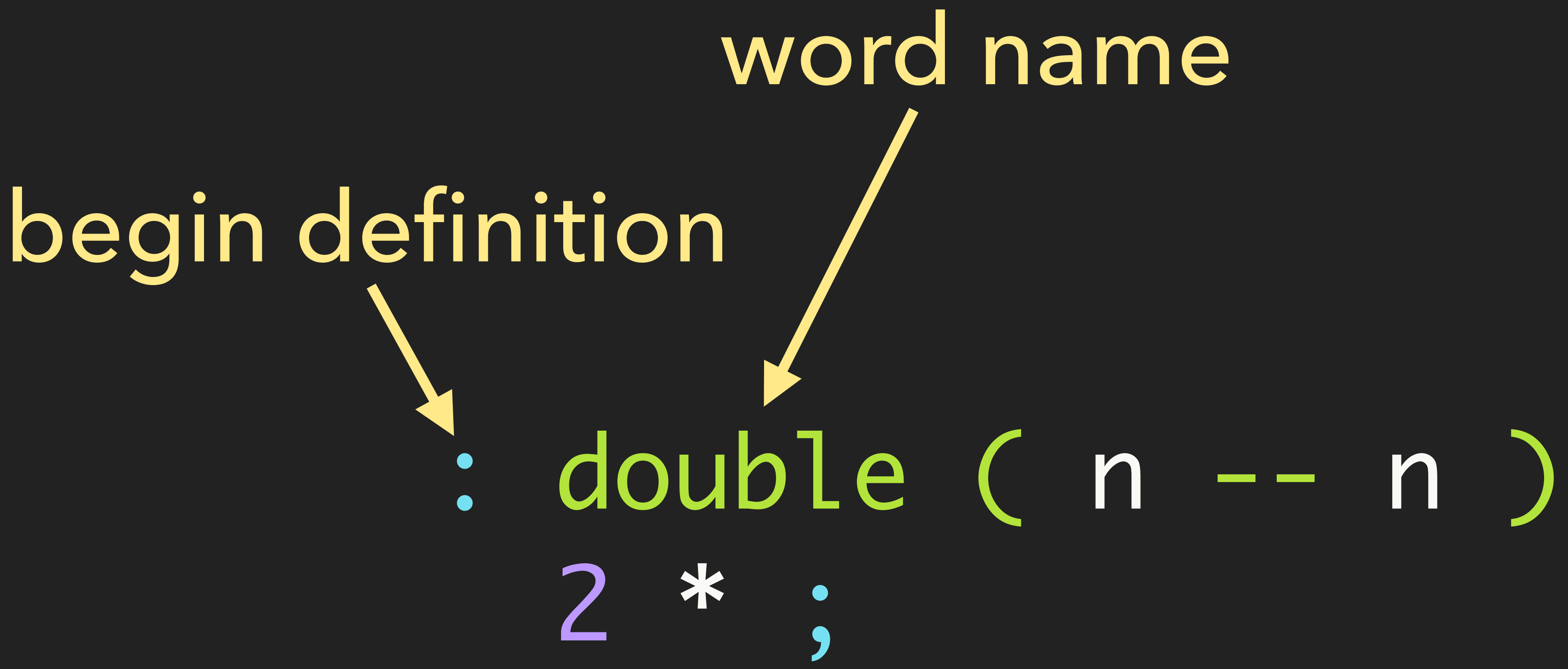
```
: double ( n -- n )
  2 * ;
```

DEFINING NEW WORDS

begin definition

```
: double ( n -- n )
  2 * ;
```

DEFINING NEW WORDS

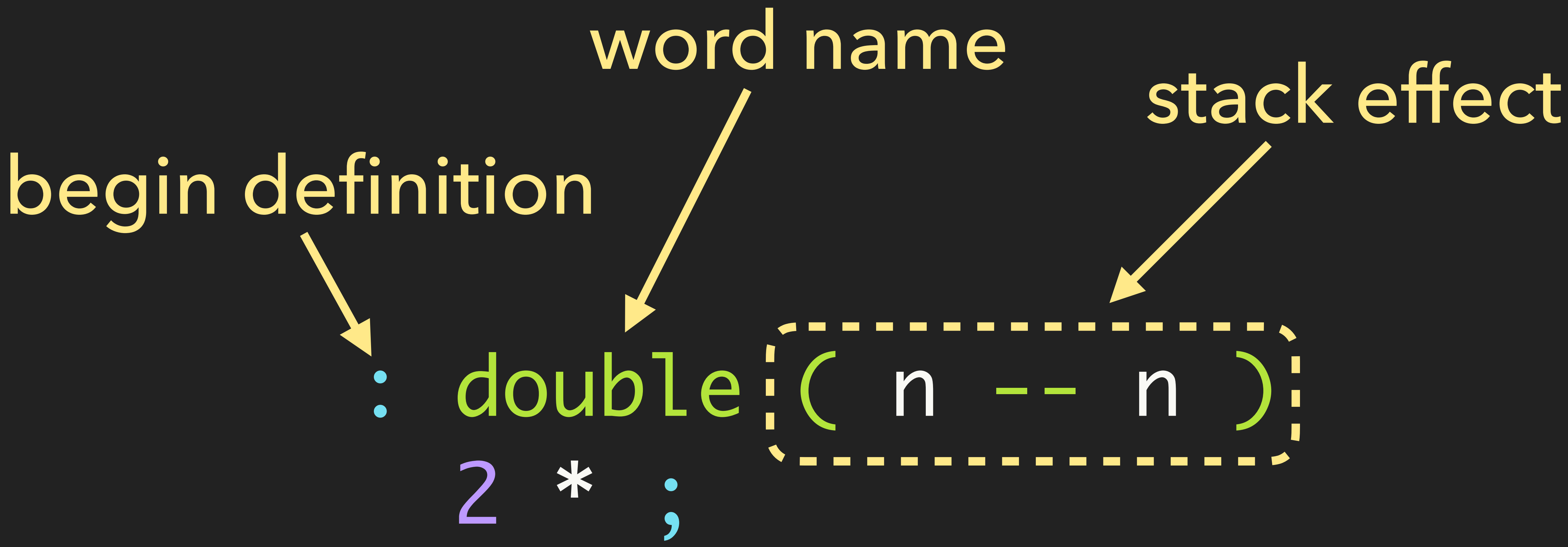


word name

begin definition

```
: double ( n -- n )
  2 * ;
```

DEFINING NEW WORDS



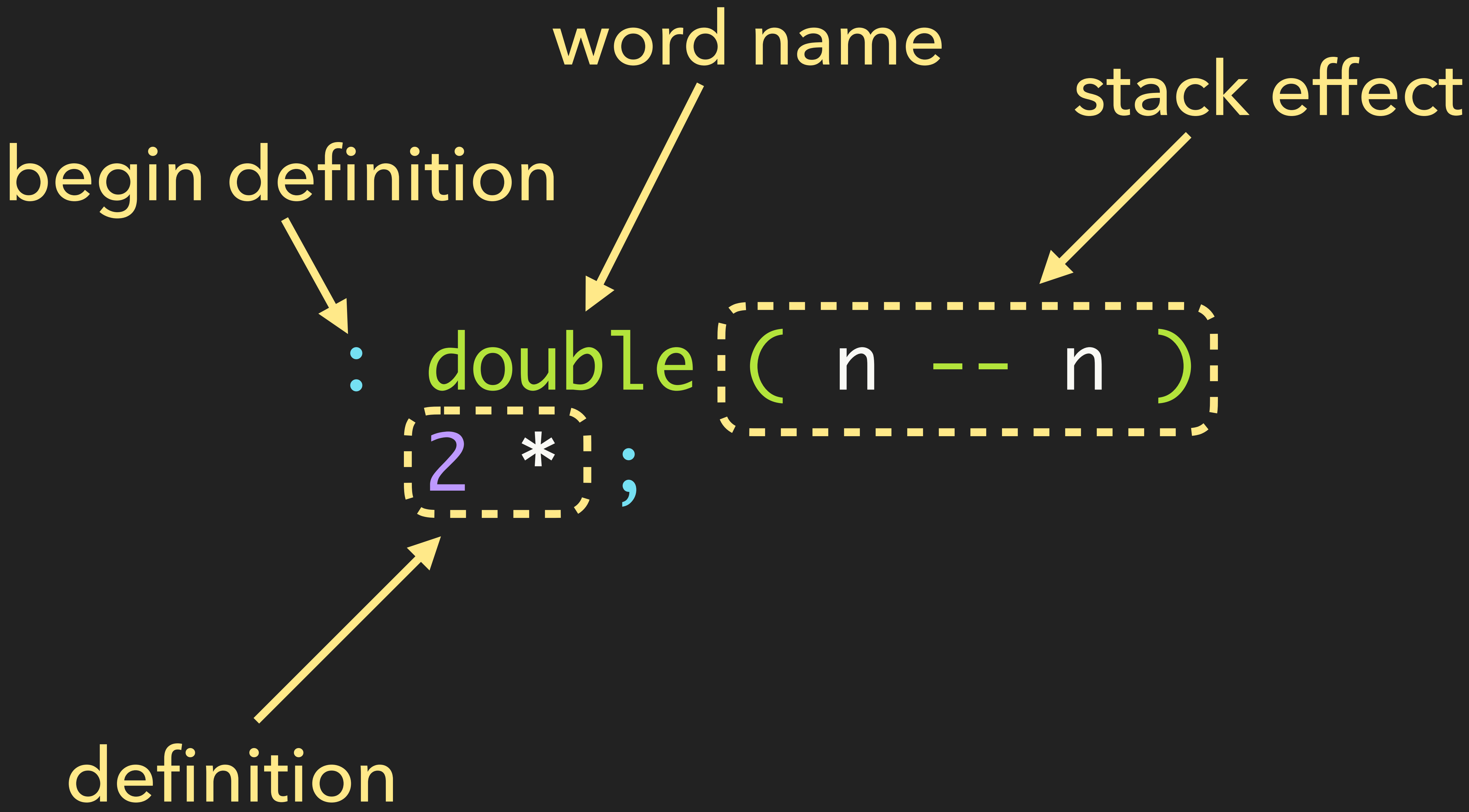
word name

stack effect

begin definition

```
: double ( n -- n )
2 * ;
```

DEFINING NEW WORDS



word name

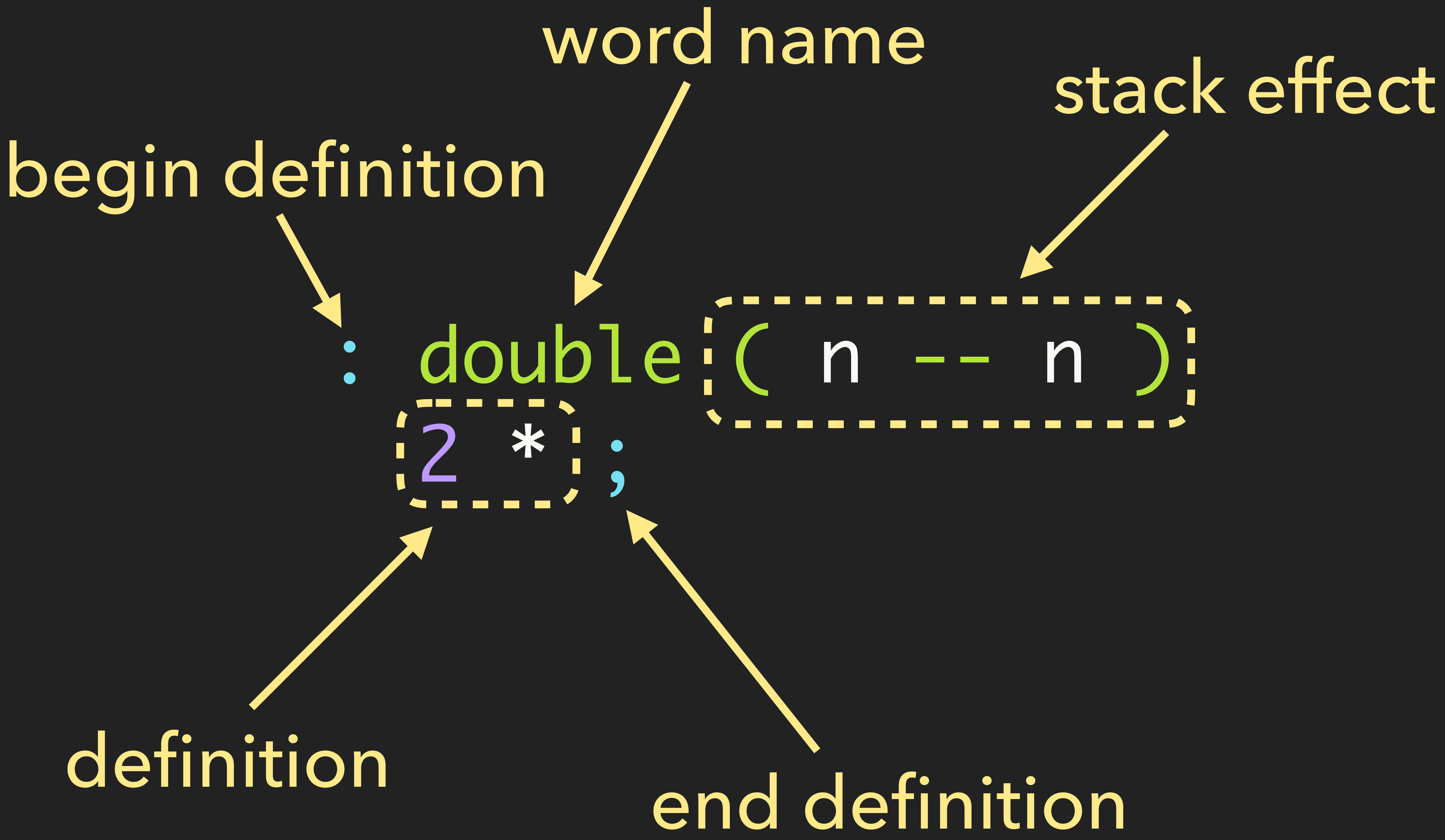
stack effect

begin definition

```
: double ( n -- n )
  2 * ;
```

definition

DEFINING NEW WORDS



word name

stack effect

begin definition

```
: double ( n -- n )
2 * ;
```

definition

end definition

DEFINING NEW WORDS

```
: double ( n -- n )
  2 * ;
```

DEFINING NEW WORDS

```
: double ( n -- n )
  2 * ;

5 double .
```

DEFINING NEW WORDS

```
: double ( n -- n )
  2 * ;

5 double .
```

(10 is output on console)

VOCABULARIES / MODULE SYSTEM

VOCABULARIES (MODULE SYSTEM)

```
USING: math ;
IN: utilities

: double ( x -- x )
  2 * ;
```

VOCABULARIES (MODULE SYSTEM)

Import Vocabularies

```
USING: math ;
IN: utilities

: double ( x -- x )
  2 * ;
```

VOCABULARIES (MODULE SYSTEM)

Import Vocabularies

```
USING: math ;
IN: utilities

: double ( x -- x )
  2 * ;
```

the multiply word exists
in the math vocab

VOCABULARIES (MODULE SYSTEM)

```
USING: math ;
IN: utilities

: double ( x -- x )
    2 * ;
```

Specify Current Vocabulary

VOCABULARIES (MODULE SYSTEM)

```
USING: math ;
IN: utilities

: double ( x -- x )
  2 * ;
```

VOCABULARIES (MODULE SYSTEM)

```
USING: math ;
IN: utilities

: double ( x -- x )
  2 * ;
```

this word gets defined
in the utilities vocabulary

VOCABULARIES (MODULE SYSTEM)

VOCABULARIES (MODULE SYSTEM)

- Vocabulary system

  implemented in Factor

- USING: and IN:

  are just words

THE BASICS

Cool Example:

```
USING: calendar calendar.english ;

: what-was-yesterday ( -- )
  1 days ago
  day-of-week day-name
  print ;
```

THE BASICS

Cool Example:

```
USING: calendar calendar.english ;

: what-was-yesterday ( -- )
  1 days ago
  day-of-week day-name
  print ;
```

Lets Step Through This Interactively

THE BASICS

What does this look like in JavaScript?

```
USING: calendar calendar.english ;

: what-was-yesterday ( -- )
  1 days ago
  day-of-week day-name
  print ;
```

THE BASICS

What does this look like in JavaScript?

```
// assume these made up functions exist
import { days, ago, dayOfWeek } from 'calendar';
import { dayName } from 'english-calendar';

function whatWasYesterday() {
  let r0 = 1;
  let r1 = days(r0);        // returns a duration object
  let r2 = ago(r1);         // return a timestamp object
  let r3 = dayOfWeek(r2);   // returns day-of-week as a number
  let r4 = dayName(r3);     // returns day-name as string
  console.log(r4);
}
```

THE BASICS

What does this look like in JavaScript?

```
// assume these made up functions exist
import { days, ago, dayOfWeek } from 'calendar';
import { dayName } from 'english-calendar';

function whatWasYesterday() {
  let r0 = 1;
  let r1 = days(r0);      // returns a duration object
  let r2 = ago(r1);       // return a timestamp object
  let r3 = dayOfWeek(r2); // returns day-of-week as a number
  let r4 = dayName(r3);   // returns day-name as string
  console.log(r4);
}
```

THE BASICS

What does this look like in JavaScript?

```
// assume these made up functions exist
import { days, ago, dayOfWeek } from 'calendar';
import { dayName } from 'english-calendar';
```

```
function whatWasYesterday() {
  let r0 = 1;
  let r1 = days(r0);      // returns a duration object
  let r2 = ago(r1);       // return a timestamp object
  let r3 = dayOfWeek(r2); // returns day-of-week as a number
  let r4 = dayName(r3);   // returns day-name as string
  console.log(r4);
}
```

THE BASICS

What does this look like in JavaScript?

```
// assume these made up functions exist
import { days, ago, dayOfWeek } from 'calendar';
import { dayName } from 'english-calendar';

function whatWasYesterday() {
  let r0 = 1;
  let r1 = days(r0);      // returns a duration object
  let r2 = ago(r1);       // return a timestamp object
  let r3 = dayOfWeek(r2); // returns day-of-week as a number
  let r4 = dayName(r3);   // returns day-name as string
  console.log(r4);
}
```

THE BASICS

What does this look like in JavaScript?

```
// assume these made up functions exist
import { days, ago, dayOfWeek } from 'calendar';
import { dayName } from 'english-calendar';

function whatWasYesterday() {
  let r0 = 1;
  let r1 = days(r0);       // returns a duration object
  let r2 = ago(r1);        // return a timestamp object
  let r3 = dayOfWeek(r2); // returns day-of-week as a number
  let r4 = dayName(r3);   // returns day-name as string
  console.log(r4);
}
```

THE BASICS

What does this look like in JavaScript?

```
// assume these made up functions exist
import { days, ago, dayOfWeek } from 'calendar';
import { dayName } from 'english-calendar';

function whatWasYesterday() {
  let r0 = 1;
  let r1 = days(r0);      // returns a duration object
  let r2 = ago(r1);       // return a timestamp object
  let r3 = dayOfWeek(r2); // returns day-of-week as a number
  let r4 = dayName(r3);   // returns day-name as string
  console.log(r4);
}
```

THE BASICS

What does this look like in JavaScript?

```
// assume these made up functions exist
import { days, ago, dayOfWeek } from 'calendar';
import { dayName } from 'english-calendar';

function whatWasYesterday() {
  let r0 = 1;
  let r1 = days(r0);      // returns a duration object
  let r2 = ago(r1);       // return a timestamp object
  let r3 = dayOfWeek(r2); // returns day-of-week as a number
  let r4 = dayName(r3);   // returns day-name as string
  console.log(r4);
}
```

THE BASICS

What does this look like in JavaScript?

```
// assume these made up functions exist
import { days, ago, dayOfWeek } from 'calendar';
import { dayName } from 'english-calendar';

function whatWasYesterday() {
  let r0 = 1;
  let r1 = days(r0);      // returns a duration object
  let r2 = ago(r1);       // return a timestamp object
  let r3 = dayOfWeek(r2); // returns day-of-week as a number
  let r4 = dayName(r3);   // returns day-name as string
  console.log(r4);
}
```

THE BASICS

What does this look like in JavaScript?

```
// assume these made up functions exist
import { days, ago, dayOfWeek } from 'calendar';
import { dayName } from 'english-calendar';

function whatWasYesterday() {
  let r0 = 1;
  let r1 = days(r0);      // returns a duration object
  let r2 = ago(r1);       // return a timestamp object
  let r3 = dayOfWeek(r2); // returns day-of-week as a number
  let r4 = dayName(r3);   // returns day-name as string
  console.log(r4);
}
```

THE BASICS

What does this look like in JavaScript?

```
// assume these made up functions exist
import { days, ago, dayOfWeek } from 'calendar';
import { dayName } from 'english-calendar';

function whatWasYesterday() {
  let r0 = 1;
  let r1 = days(r0);      // returns a duration object
  let r2 = ago(r1);       // return a timestamp object
  let r3 = dayOfWeek(r2); // returns day-of-week as a number
  let r4 = dayName(r3);   // returns day-name as string
  console.log(r4);
}
```

Variables!

THE BASICS

What does this look like in JavaScript?

```
// assume these made up functions exist
import { days, ago, dayOfWeek } from 'calendar';
import { dayName } from 'english-calendar';

function whatWasYesterday() {
  let r0 = 1;
  let r1 = days(r0);       // returns a duration object
  let r2 = ago(r1);        // return a timestamp object
  let r3 = dayOfWeek(r2); // returns day-of-week as a number
  let r4 = dayName(r3);   // returns day-name as string
  console.log(r4);
}
```

Lets remove all them Variables!

THE BASICS

What does this look like in JavaScript?

```
import { days, ago, dayOfWeek } from 'calendar';
import { dayName } from 'english-calendar';

function whatWasYesterday() {
  console.log(
    dayName(
      dayOfWeek(
        ago(
          days(
            1
          )
        )
      )
    )
  )
}
```

THE BASICS

What does this look like in JavaScript?

```
import { days, ago, dayOfWeek } from 'calendar';
import { dayName } from 'english-calendar';

function whatWasYesterday() {
  console.log(
    dayName(
      dayOfWeek(
        ago(
          days(
            1
          )
        )
      )
    )
  )
}
```

JS Call Nesting

THE BASICS

What does this look like in JavaScript?

```
import { days, ago, dayOfWeek } from 'calendar';
import { dayName } from 'english-calendar';

function whatWasYesterday() {
  console.log(
    dayName(
      dayOfWeek(
        ago(
          days(
            1
          )
        )
      )
    )
  )
}
```

Factor Word Definition

```
USING: calendar calendar.english ;

: what-was-yesterday ( -- )
  1 days ago
  day-of-week day-name
  print ;
```

THE BASICS

What does this look like in JavaScript?

```
import { days, ago, dayOfWeek } from 'calendar';
import { dayName } from 'english-calendar';

function whatWasYesterday() {
  console.log(
    dayName(
      dayOfWeek(
        ago(
          days(
            1
          )
        )
      )
    )
  )
}
```

Factor Word Definition

```
USING: calendar calendar.english ;

: what-was-yesterday ( -- )
  1 days ago
  day-of-week day-name
  print ;
```

In Stack Based Languages
Sequence of Words = Function Composition

THE BASICS

Name Operations

Not Variables

THE BASICS

Lets refactor this…

```
USING: calendar calendar.english ;

: what-was-yesterday ( -- )
  1 days ago
  day-of-week day-name
  print ;
```

THE BASICS

Lets refactor this…

```
USING: calendar calendar.english ;

: what-was-yesterday ( -- )
  1 days ago
  day-of-week day-name
  print ;
```

yesterday

THE BASICS

Lets refactor this…

```
USING: calendar calendar.english ;

: what-was-yesterday ( -- )
  1 days ago
  day-of-week day-name
  print ;
```

→ print-day-name

THE BASICS

After refactoring…

```
USING: calendar calendar.english ;

: yesterday ( -- timestamp )
  1 days ago ;

: print-day-name ( timestamp -- )
  day-of-week day-name
  print ;

: what-was-yesterday ( -- )
  yesterday print-day-name ;
```

THE BASICS

After refactoring…

```
USING: calendar calendar.english ;

: yesterday ( -- timestamp )
  1 days ago ;

: print-day-name ( timestamp -- )
  day-of-week day-name
  print ;

: what-was-yesterday ( -- )
  yesterday print-day-name ;
```

SHUFFLE WORDS

```
: sq ( n -- n )
```

SHUFFLE WORDS

```
: sq ( n -- n )
  whats the definition ?
```

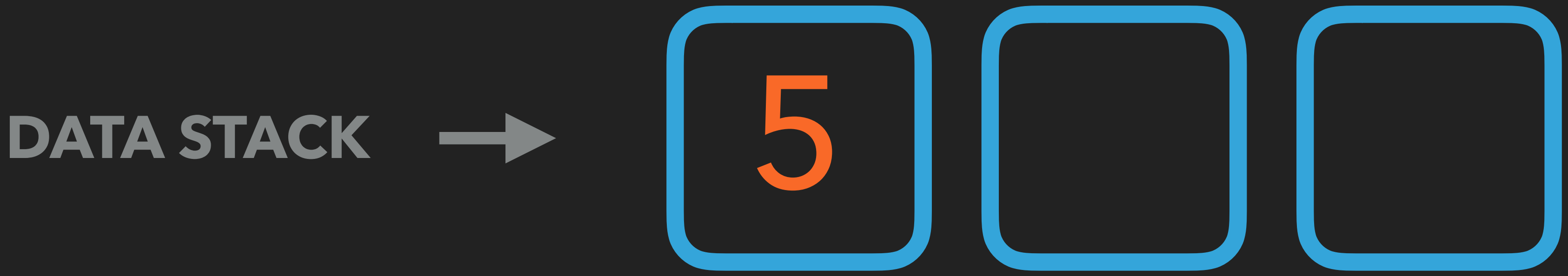
SHUFFLE WORDS

```
: sq ( n -- n )
  dup * ;
```

SHUFFLE WORDS

```
: sq ( n -- n )
  dup * ;
```

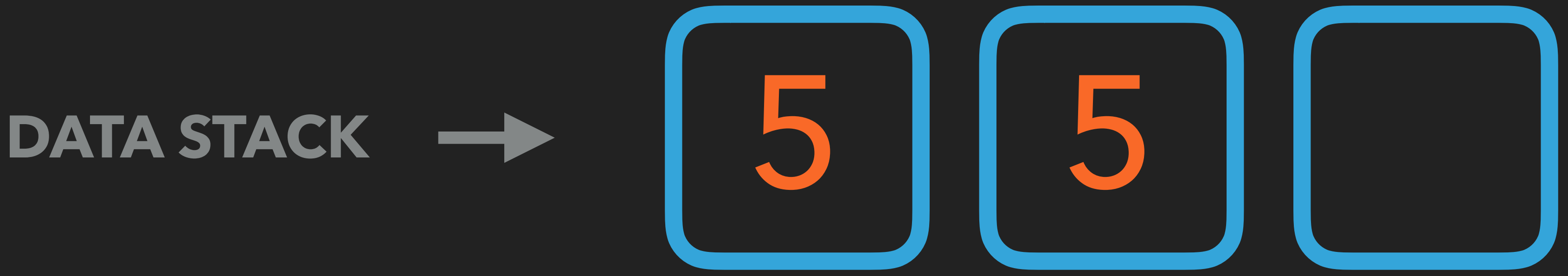
```
dup ( x -- x x )
```



SHUFFLE WORDS

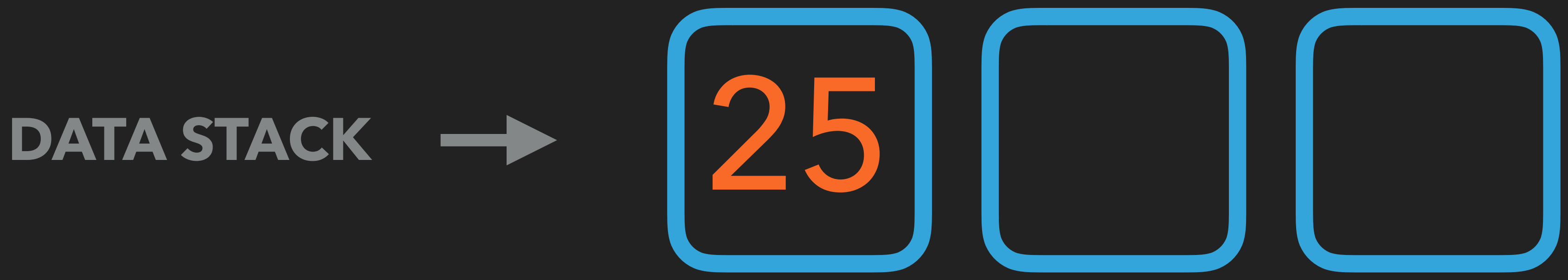
```
: sq ( n -- n )
  dup * ;
```

```
* ( x y -- z )
```



SHUFFLE WORDS

```
: sq ( n -- n )
  dup * ;
```



SHUFFLE WORDS

```
: sq ( n -- n )
```

SHUFFLE WORDS

```
: sq ( n -- n )
  dup * ;
```

Word	Stack Effect
dup	(x -- x x)

SHUFFLE WORDS

```
: sq ( n -- n )
  dup * ;
```

Word	Stack Effect
dup	(x -- x x)
swap	(x y -- y x)
drop	(x --)
over	(x y -- x y x)
nip	(x y -- y)

SHUFFLE WORDS

```
: sq ( n -- n )
  dup * ;
```

Word	Stack Effect
dup	(x -- x x)
swap	(x y -- y x)
drop	(x --)
over	(x y -- x y x)
nip	(x y -- y)

SHUFFLE WORDS

```
: sq ( n -- n )
  dup * ;
```

Word	Stack Effect
dup	(x -- x x)
swap	(x y -- y x)
drop	(x --)
over	(x y -- x y x)
nip	(x y -- y)

SHUFFLE WORDS

```
: sq ( n -- n )
  dup * ;
```

Word	Stack Effect
dup	(x -- x x)
swap	(x y -- y x)
drop	(x --)
over	(x y -- x y x)
nip	(x y -- y)

SHUFFLE WORDS

```
: sq ( n -- n )
  dup * ;
```

Word	Stack Effect
dup	(x -- x x)
swap	(x y -- y x)
drop	(x --)
over	(x y -- x y x)
nip	(x y -- y)

THE BASICS

Multiple Return Values

```
/mod    ( x y -- q r )
```

pronounced “slashmod”

THE BASICS

Multiple Return Values

```
/mod   ( x y -- q r )
```

Divides x by y leaving quotient q and remainder r on the stack

THE BASICS

Multiple Return Values

```
/mod   ( x y -- q r )
```

```
7 3 /mod
```

THE BASICS

Multiple Return Values

```
/mod   ( x y -- q r )
```

```
7 3 /mod
```

```
( leaves 2 and 1 on the data stack )
```

THE BASICS

Multiple Return Values

```
/mod   ( x y -- q r )
```

```
7 3 /mod
```

(leaves 2 and 1 on the data stack)

```
: quotient ( x y -- q )
  /mod drop ;
```

THE BASICS

Multiple Return Values

```
/mod   ( x y -- q r )
```

```
7 3 /mod
```

(leaves 2 and 1 on the data stack)

```
: quotient ( x y -- q )
  /mod drop ;
```

↑

(r --)

THE BASICS

Multiple Return Values

```
/mod  ( x y -- q r )
```

```
7 3 /mod
```

(leaves 2 and 1 on the data stack)

```
: quotient ( x y -- q )
  /mod drop ;
```

```
: remainder ( x y -- r )
  /mod nip ;
```

THE BASICS

Multiple Return Values

```
/mod  ( x y -- q r )
```

```
7 3 /mod
```

(leaves 2 and 1 on the data stack)

```
: quotient ( x y -- q )
  /mod drop ;
```

```
: remainder ( x y -- r )
  /mod nip ;
```

```
( q r -- r )
```

LITERALS

LITERALS

- Booleans `t` (true) `f` (false)
- Numbers `123` `1.25` `22/7`
- Strings `"factor"`

LITERALS

LITERALS

Arrays

```
{ 1 2 3 }
```

LITERALS

Arrays

```
{ 1 2 3 } sum .
```

(seq -- sum)

LITERALS

Arrays

```
{ 1 2 3 } sum .
```

(outputs 6)

LITERALS

Arrays

```
{ 1 2 3 } sum .
```

(outputs 6)

Hashtables

```
H{
   { "a" 10 }
   { "b" 20 }
}
```

LITERALS

Arrays

```
{ 1 2 3 } sum .
```

(outputs 6)

Hashtables

```
"a" H{
        { "a" 10 }
        { "b" 20 }
    } at .
```

(key hashtable -- value)

LITERALS

Arrays

```
{ 1 2 3 } sum .
```

(outputs 6)

Hashtables

```
"a" H{
      { "a" 10 }
      { "b" 20 }
    } at .
```

(outputs 10)

LITERALS

Arrays

```
{ 1 2 3 } sum .
```

(outputs 6)

Hashtables

```
"a" H{
      { "a" 10 }
      { "b" 20 }
    } at .
```

(outputs 10)

LITERALS

Quotations

```
[ 12 23 + . ]
```

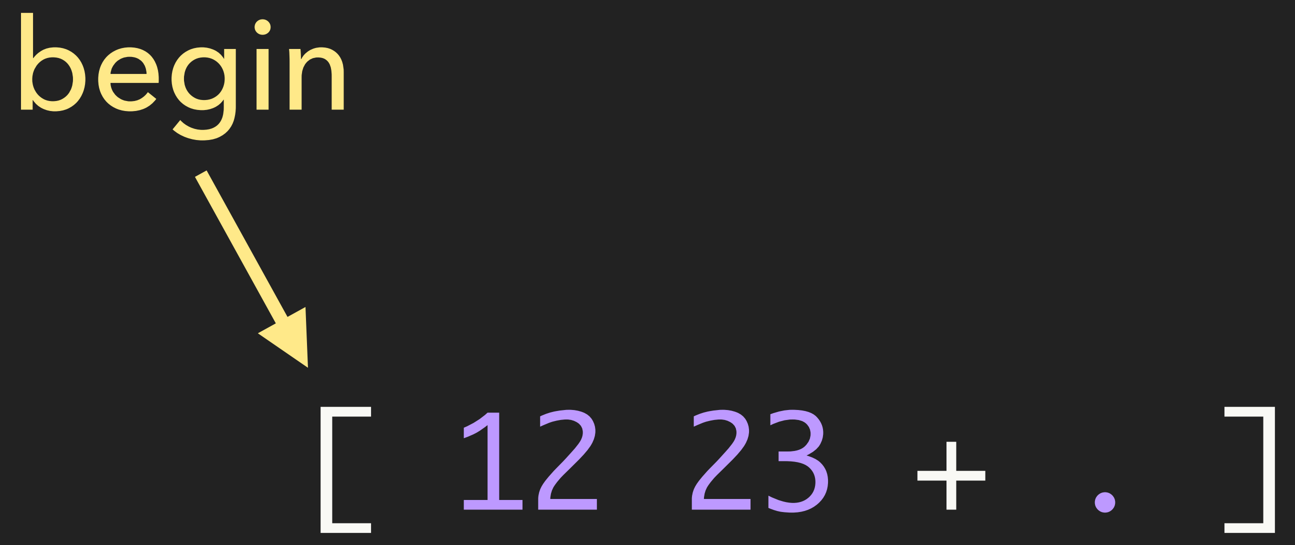
LITERALS

Quotations (anonymous functions)

```
[ 12 23 + . ]
```

LITERALS

Quotations (anonymous functions)

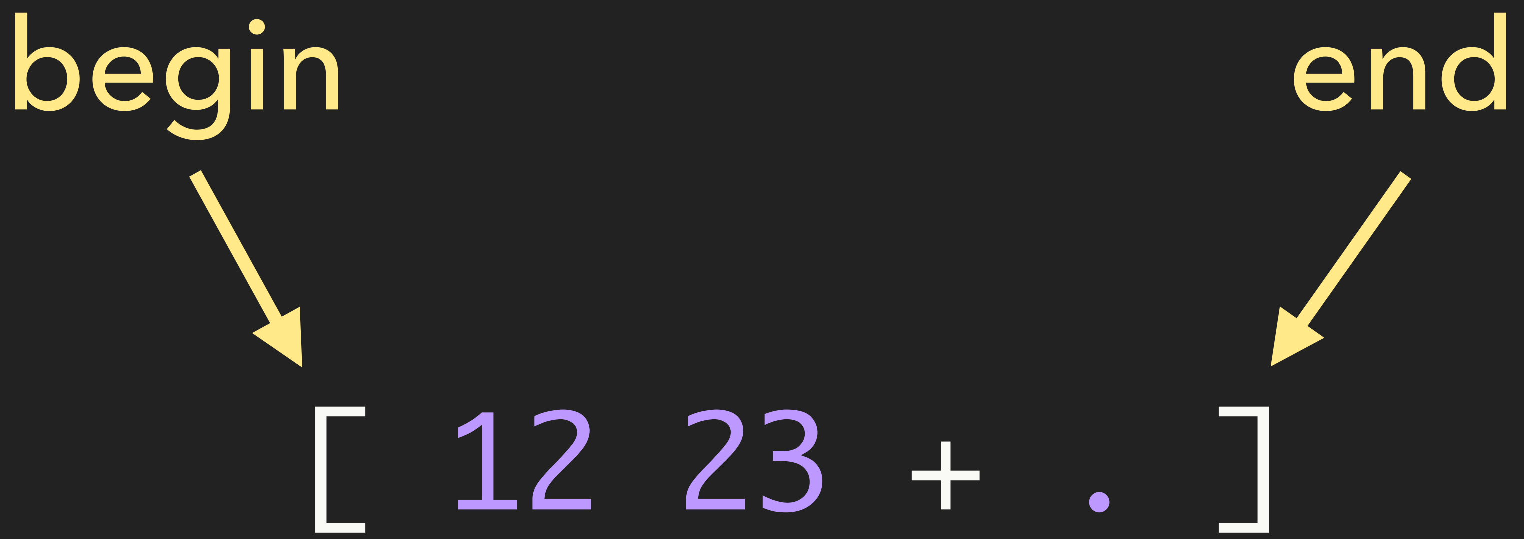


begin

```
[ 12 23 + . ]
```

LITERALS

Quotations (anonymous functions)



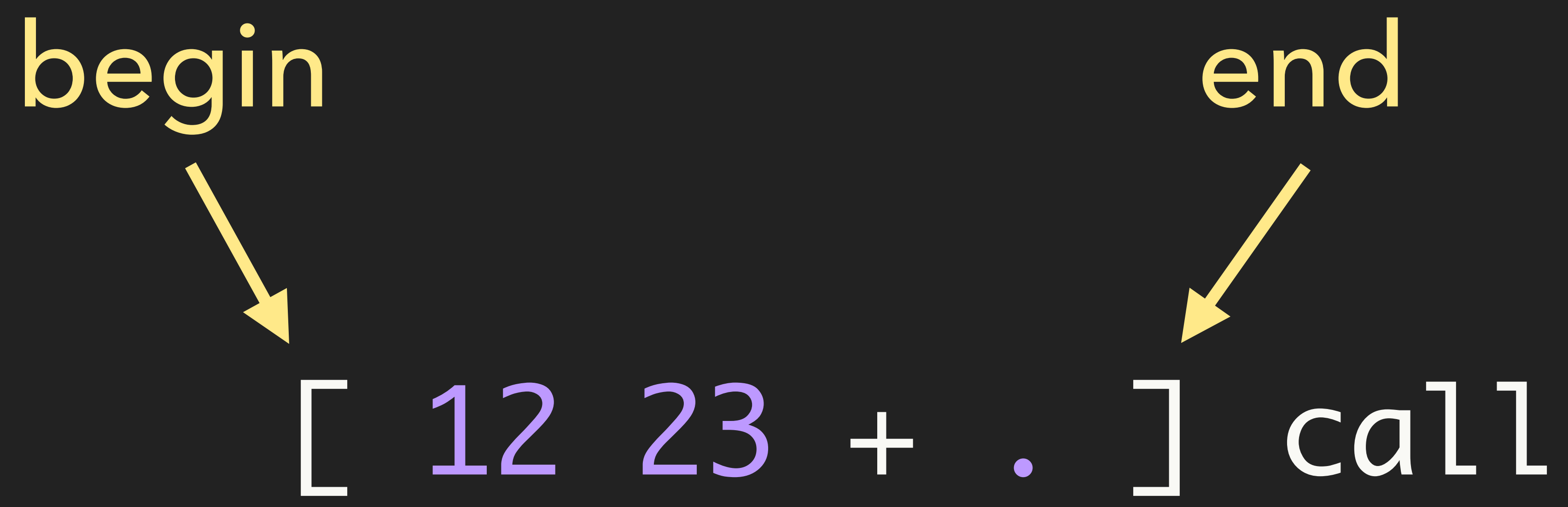
begin

end

```
[ 12 23 + . ]
```

LITERALS

Quotations (anonymous functions)



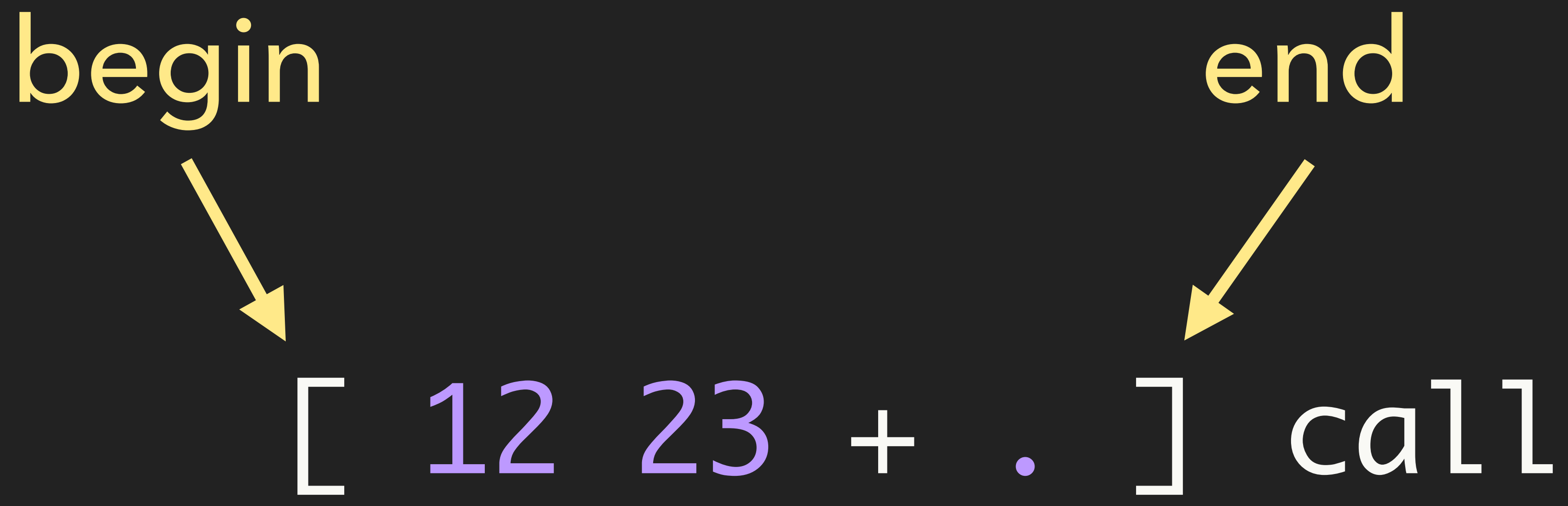
begin

end

```
[ 12 23 + . ] call
```

LITERALS

Quotations (anonymous functions)



begin

end

```
[ 12 23 + . ] call
```

(outputs 35)

COMBINATORS

COMBINATORS

- Words that take Quotations as input

COMBINATORS

- Words that take Quotations as input
- Example: if
  - takes a boolean value
  - and a quotation for the true case
  - and a quotation for the false case
  - evaluates appropriate quotation

COMBINATORS

- Words that take Quotations as input
- Example: if
  - takes a boolean value
  - and a quotation for the true case
  - and a quotation for the false case
  - evaluates appropriate quotation
- Stack effect: (..a ? true: (..a -- ..b)   false: (..a -- ..b)   --   ..b)

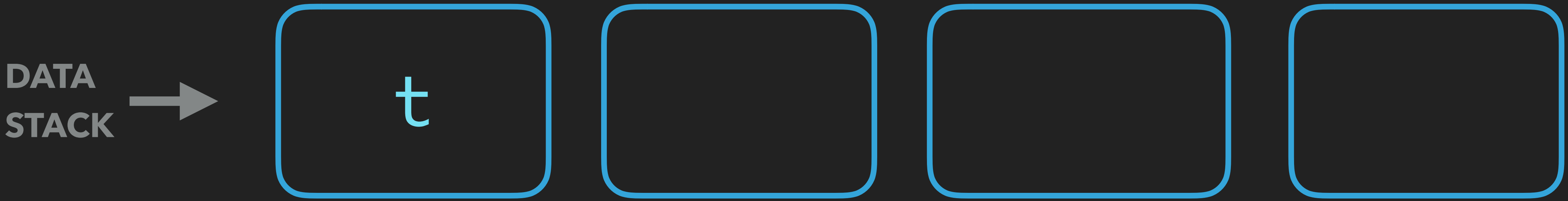
COMBINATORS

```
t [ "Yup" ] [ "Nope" ] if print
```

DATA STACK

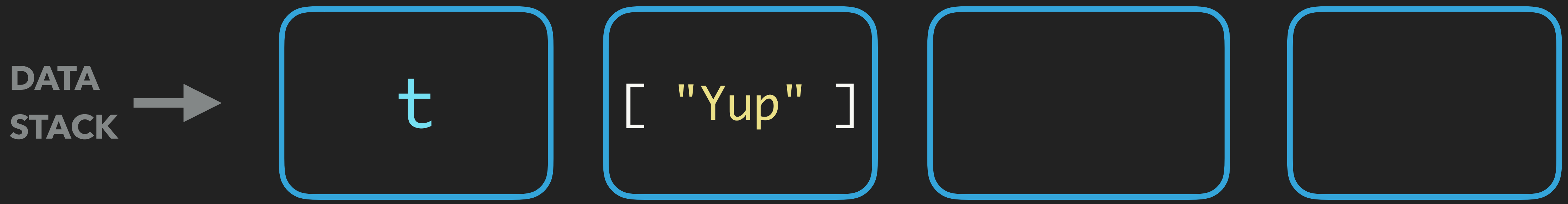
COMBINATORS

```
t [ "Yup" ] [ "Nope" ] if print
```



COMBINATORS

```
t [ "Yup" ] [ "Nope" ] if print
```



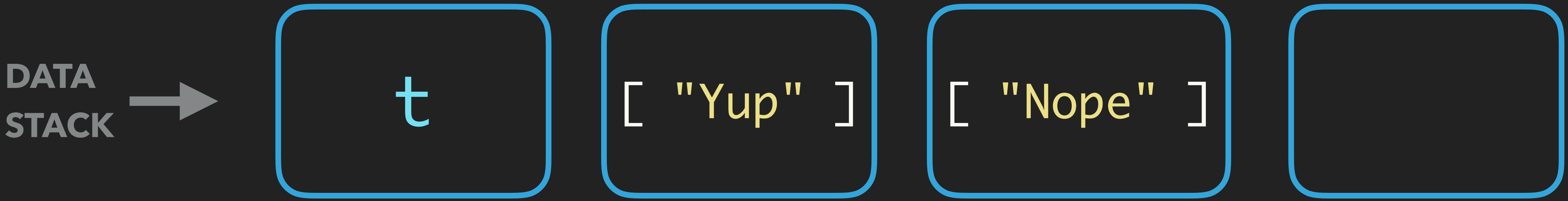
DATA STACK

t

["Yup"]

COMBINATORS

```
t [ "Yup" ] [ "Nope" ] if print
```



DATA STACK

t

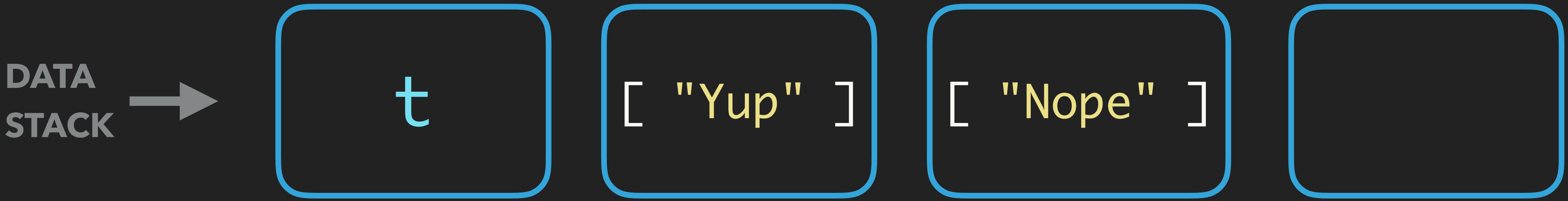
["Yup"]

["Nope"]

COMBINATORS

```
t [ "Yup" ] [ "Nope" ] if print
```

```
( ..a ? true: ( ..a -- ..b ) false: ( ..a -- ..b ) -- ..b )
```



DATA STACK

t

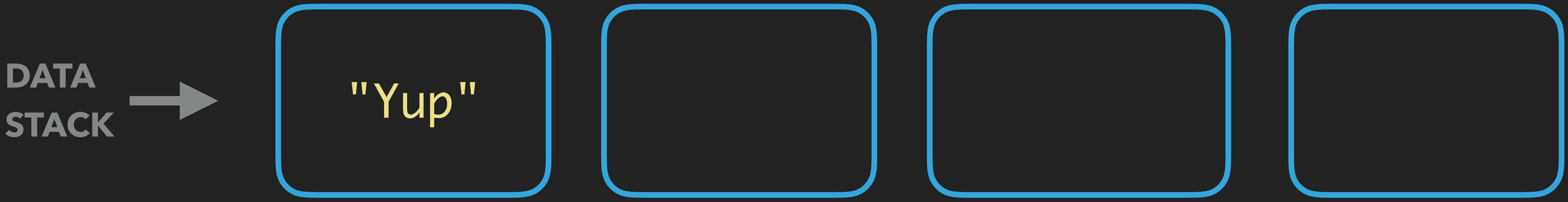
["Yup"]

["Nope"]

COMBINATORS

```
t [ "Yup" ] [ "Nope" ] if print
```

```
( obj -- )
```



DATA STACK

"Yup"

COMBINATORS

```
t [ "Yup" ] [ "Nope" ] if print
```

(outputs Yup)

COMBINATOR EXAMPLES

COMBINATOR EXAMPLES

times (… n quot: (… – …) – …)

```
10 [
  "Factor is terse!" print
] times
```

COMBINATOR EXAMPLES

times (… n quot: (… – …) – …)

```
10 [
  "Factor is terse!" print
] times
```

while (..a pred: (..a – ..b ?) body: (..b – ..a) – ..a)

```
1 [ dup 10 <= ] [
  dup sq . 1 +
] while drop
```

COMBINATOR EXAMPLES

times (… n quot: (… – …) – …)

```
10 [
   "Factor is terse!" print
] times
```

while (..a pred: (..a – ..b ?) body: (..b – ..a) – ..a)

```
1 [ dup 10 <= ] [
   dup sq . 1 +
] while drop
```

COMBINATOR EXAMPLES

times (… n quot: (… – …) – …)

```
10 [
   "Factor is terse!" print
] times
```

while (..a pred: (..a – ..b ?) body: (..b – ..a) – ..a)

```
1 [ dup 10 <= ] [
   dup sq . 1 +
] while drop
```

COMBINATOR EXAMPLES

times (… n quot: (… – …) – …)

```
10 [
   "Factor is terse!" print
] times
```

while (..a pred: (..a – ..b ?) body: (..b – ..a) – ..a)

```
1 [ dup 10 <= ] [
   dup sq . 1 +
] while drop
```

CONTROL FLOW WORDS

CONTROL FLOW WORDS

- if, when, unless, while, until, times

  control flow words

  implemented as combinators

- Create your own control flow words

  by combining existing primitives

COMBINATOR EXAMPLES (SEQUENCES)

map (… seq quot: (… x – … mx) – … newseq)

filter (… seq quot: (… x – … ?) – … newseq)

```
: sum-of-squares-of-evens ( seq -- sum )
  [ even? ] filter
  [ sq ] map
  sum ;
```

COMBINATOR EXAMPLES (SEQUENCES)

map   (… seq quot: (… x – … mx) – … newseq)

filter   (… seq quot: (… x – … ?) – … newseq)

```
: sum-of-squares-of-evens ( seq -- sum )
  [ even? ] filter
  [ sq ] map
  sum ;
```

Lets Step Through This Interactively

COMBINATOR EXAMPLES

dip	bi	bi*	bi@
2dip	2bi	2bi*	2bi@
keep	tri	tri*	tri@
2keep	2tri	2tri*	2tri@
curry	2curry	compose	with

COMBINATOR EXAMPLES (SEQUENCE OPERATIONS)

each each-index

map map-index reduce

filter partition

2each 2map 3each 3map

any? all?

COMBINATOR EXAMPLES

And Many Many More

LOCAL VARIABLES

LOCALS

Area of a triangle using Herons formula

LOCALS

Area of a triangle using Herons formula

```
! given length of sides: a b c
! area = sqrt(p * p-a * p-b * p-c)
! where p = (a + b + c)/2 is perimeter
```

LOCALS

Area of a triangle using Herons formula

```
! given length of sides: a b c
! area = sqrt(p * p-a * p-b * p-c)
! where p = (a + b + c)/2 is perimeter
: triangle-area ( a b c -- area )
  3dup + + 2 /                ! a b c p
  [ swap - ] keep [ rot ] dip ! b p-c a p
  [ swap - ] keep [ rot ] dip ! p-c p-b a p
  [ swap - ] keep             ! p-c p-b p-a p
  * * * sqrt ;
```

LOCALS

Area of a triangle using Herons formula

```
! given length of sides: a b c
! area = sqrt(p * p-a * p-b * p-c)
! where p = (a + b + c)/2 is perimeter
: triangle-area ( a b c -- area )
  3dup + + 2 /                ! a b c p
  [ swap - ] keep [ rot ] dip ! b p-c a p
  [ swap - ] keep [ rot ] dip ! p-c p-b a p
  [ swap - ] keep             ! p-c p-b p-a p
  * * * sqrt ;
```

Too much Stack Manipulation!!

LOCALS

Area of a triangle using Herons formula

(with locals)

LOCALS

Area of a triangle using Herons formula
(with locals)

```
! area = sqrt(p * p-a * p-b * p-c)
! where p = (a + b + c)/2 is perimeter
:: triangle-area ( a b c -- area )
  a b + c + 2 / :> p
  p a - :> p-a
  p b - :> p-b
  p c - :> p-c
  p p-a * p-b * p-c *
  sqrt ;
```

LOCALS

Area of a triangle using Herons formula
(with locals)

```
! area = sqrt(p * p-a * p-b * p-c)
! where p = (a + b + c)/2 is perimeter
:: triangle-area ( a b c -- area )
   a b + c + 2 / :> p
   p a - :> p-a
   p b - :> p-b
   p c - :> p-c
   p p-a * p-b * p-c *
   sqrt ;
```

Define word with local variables

LOCALS

Area of a triangle using Herons formula
(with locals)

```
! area = sqrt(p * p-a * p-b * p-c)
! where p = (a + b + c)/2 is perimeter
:: triangle-area ( a b c -- area )
  a b + c + 2 / :> p
  p a - :> p-a
  p b - :> p-b
  p c - :> p-c
  p p-a * p-b * p-c *
  sqrt ;
```

Named local variables

LOCALS

Area of a triangle using Herons formula
(with locals)

```
! area = sqrt(p * p-a * p-b * p-c)
! where p = (a + b + c)/2 is perimeter
:: triangle-area ( a b c -- area )
  a b + c + 2 / :> p
  p a - :> p-a
  p b - :> p-b
  p c - :> p-c
  p p-a * p-b * p-c *
  sqrt ;
```

LOCALS

Area of a triangle using Herons formula (with locals)

```
! area = sqrt(p * p-a * p-b * p-c)
! where p = (a + b + c)/2 is perimeter
:: triangle-area ( a b c -- area )
  a b + c + 2 / :> p
  p a - :> p-a
  p b - :> p-b
  p c - :> p-c
  p p-a * p-b * p-c *
  sqrt ;
```

Assignment operator

LOCALS

Area of a triangle using Herons formula (with locals)

```
! area = sqrt(p * p-a * p-b * p-c)
! where p = (a + b + c)/2 is perimeter
:: triangle-area ( a b c -- area )
  a b + c + 2 / :> p
  p a - :> p-a
  p b - :> p-b
  p c - :> p-c
  p p-a * p-b * p-c *
  sqrt ;
```

} a lot more readable

LOCALS

Area of a triangle using Herons formula
(with locals)

```
! area = sqrt(p * p-a * p-b * p-c)
! where p = (a + b + c)/2 is perimeter
:: triangle-area ( a b c -- area )
  a b + c + 2 / :> p
  p a - :> p-a
  p b - :> p-b
  p c - :> p-c
  p p-a * p-b * p-c *
  sqrt ;
```

LOCALS

LOCALS

- :: and :>

  are words

  implemented as syntax extensions

  in the locals vocabulary

- Less than 1%

  of words in the core distribution

  use Local Variables

OBJECT ORIENTATION

OBJECT ORIENTATION IN FACTOR

OBJECT ORIENTATION IN FACTOR

- Tuples: collection of named slots
- Auto-generated slot accessor words
- Generic words dispatch on tuple type
- Methods implement generic word for specific Tuple

OBJECT ORIENTATION IN FACTOR

```
TUPLE: rectangle length breath ;

length>> ( obj -- value )
breath>> ( obj -- value )

>>length ( obj value -- obj )
>>breath ( obj value -- obj )
```

OBJECT ORIENTATION IN FACTOR

```
TUPLE: rectangle length breath ;

length>> ( obj -- value )
breath>> ( obj -- value )

>>length ( obj value -- obj )
>>breath ( obj value -- obj )
```

OBJECT ORIENTATION IN FACTOR

```
TUPLE: rectangle length breath ;

length>> ( obj -- value )
breath>> ( obj -- value )

>>length ( obj value -- obj )
>>breath ( obj value -- obj )
```

OBJECT ORIENTATION IN FACTOR

```
TUPLE: rectangle length breath ;

length>> ( obj -- value )
breath>> ( obj -- value )

>>length ( obj value -- obj )
>>breath ( obj value -- obj )
```

Getter Words

OBJECT ORIENTATION IN FACTOR

```
TUPLE: rectangle length breath ;

length>> ( obj -- value )
breath>> ( obj -- value )

>>length ( obj value -- obj )
>>breath ( obj value -- obj )
```

OBJECT ORIENTATION IN FACTOR

```
TUPLE: rectangle length breath ;

length>> ( obj -- value )
breath>> ( obj -- value )

>>length ( obj value -- obj )
>>breath ( obj value -- obj )
```

Setter Words

OBJECT ORIENTATION IN FACTOR

```
TUPLE: rectangle length breath ;
TUPLE: circle radius ;

GENERIC: area ( obj -- n )

M: rectangle area
  [ length>> ] [ breath>> ]
  bi * ;

M: circle area
  radius>> sq
  3.14 * ;
```

OBJECT ORIENTATION IN FACTOR

```
TUPLE: rectangle length breath ;
TUPLE: circle radius ;

GENERIC: area ( obj -- n )

M: rectangle area
  [ length>> ] [ breath>> ]
  bi * ;

M: circle area
  radius>> sq
  3.14 * ;
```

OBJECT ORIENTATION IN FACTOR

```
TUPLE: rectangle length breath ;
TUPLE: circle radius ;

GENERIC: area ( obj -- n )

M: rectangle area
  [ length>> ] [ breath>> ]
  bi * ;

M: circle area
  radius>> sq
  3.14 * ;
```

OBJECT ORIENTATION IN FACTOR

```
TUPLE: rectangle length breath ;
TUPLE: circle radius ;

GENERIC: area ( obj -- n )

M: rectangle area
  [ length>> ] [ breath>> ]
  bi * ;

M: circle area
  radius>> sq
  3.14 * ;
```

Method Definitions

OBJECT ORIENTATION IN FACTOR

```
TUPLE: rectangle length breath ;
TUPLE: circle radius ;

GENERIC: area ( obj -- n )

M: rectangle area
  [ length>> ] [ breath>> ]
  bi * ;

M: circle area
  radius>> sq
  3.14 * ;
```

OBJECT ORIENTATION IN FACTOR

```
TUPLE: rectangle length breath ;
TUPLE: circle radius ;

GENERIC: area ( obj -- n )

M: rectangle area
  [ length>> ] [ breath>> ]
  bi * ;

M: circle area
  radius>> sq
  3.14 * ;
```

OBJECT ORIENTATION IN FACTOR

```
: <rectangle> ( l b -- rect )
  rectangle new
  swap >>breath
  swap >>length ;

: <circle> ( r -- circle )
  circle new
  swap >>radius ;
```

OBJECT ORIENTATION IN FACTOR

```
: <rectangle> ( l b -- rect )
  rectangle new
  swap >>breath
  swap >>length ;

: <circle> ( r -- circle )
  circle new
  swap >>radius ;
```

Constructor Words

OBJECT ORIENTATION IN FACTOR

```
: <rectangle> ( l b -- rect )
  rectangle new
  swap >>breath
  swap >>length ;

: <circle> ( r -- circle )
  circle new
  swap >>radius ;
```

OBJECT ORIENTATION IN FACTOR

```
: <rectangle> ( l b -- rect )
  rectangle new
  swap >>breath
  swap >>length ;

: <circle> ( r -- circle )
  circle new
  swap >>radius ;
```

OBJECT ORIENTATION IN FACTOR

```
10 5 <rectangle> area .
```

OBJECT ORIENTATION IN FACTOR

```
10 5 <rectangle> area .
```

(outputs 50)

OBJECT ORIENTATION IN FACTOR

```
10 5 <rectangle> area .
```

(outputs 50)

```
10 <circle> area .
```

OBJECT ORIENTATION IN FACTOR

```
10 5 <rectangle> area .
```

(outputs 50)

```
10 <circle> area .
```

(outputs 314)

OBJECT ORIENTATION IN FACTOR

OBJECT ORIENTATION IN FACTOR

- TUPLE: GENERIC: M:

  are words

  implemented as syntax extensions

SYNTAX RECAP

```
:                                   ::

“

(
                                    M:
{

H{

[                                TUPLE:
```

SYNTAX RECAP

:

Define New Word

::

“

(

```
USING: effects.parser words ;
IN: syntax
SYNTAX: : (:) define-declared ;
```

M:

{

H{

[

TUPLE:

SYNTAX RECAP

String Literal

:

“

(

{

H{

[

::

M:

TUPLE:

```
USING: sequences strings.parser ;
IN: syntax
SYNTAX: " parse-string suffix! ;
```

SYNTAX RECAP

:

Hashtable Literal

::

“

(

```
USING: hashtables parser ;
IN: syntax
SYNTAX: H{ \ } [ parse-hashtable ]
parse-literal ;
```

M:

{

H{

[

TUPLE:

SYNTAX RECAP

Define a New Tuple

:

“

(

{

H{

[

::

```
USING: classes.tuple classes.tuple.parser ;
IN: syntax
SYNTAX: TUPLE: parse-tuple-definition
define-tuple-class ;
```

M:

TUPLE:

SYNTAX RECAP

```
:                         ::

“

(
                          M:
{

H{

[                     TUPLE:
```

SYNTAX RECAP

:

::

“

Syntax == Words

(

M:

{

H{

[

TUPLE:

SYNTAX RECAP

`:`

`::`

`“`

Syntax == Words

`(`

defined

`M:`

`{`

in Factor

`H{`

`[`

`TUPLE:`

SYNTAX RECAP

SYNTAX:

SYNTAX RECAP

`SYNTAX:`	defines a new parsing word (i.e. syntax extension)

SYNTAX RECAP

SYNTAX: defines a new parsing word (i.e. syntax extension)

```
USING: bootstrap.syntax parser words ;
IN: syntax
SYNTAX: SYNTAX:
    scan-new-word mark-top-level-syntax parse-definition
    define-syntax ;
```

SYNTAX RECAP

SYNTAX:

defines a new parsing word
(i.e. syntax extension)

```
USING: bootstrap.syntax parser words ;
IN: syntax
SYNTAX: SYNTAX:
    scan-new-word mark-top-level-syntax parse-definition
    define-syntax ;
```

SYNTAX RECAP

SYNTAX: defines a new parsing word (syntax extension)

```
USING: bootstrap.syntax parser words ;
IN: syntax
SYNTAX: SYNTAX:
    scan-new-word mark-top-level-syntax parse-definition
    define-syntax ;
```

(Factor is self hosted)

SYNTAX RECAP

SYNTAX: defines a new parsing word
(syntax extension)

```
USING: bootstrap.syntax parser words ;
IN: syntax
SYNTAX: SYNTAX:
    scan-new-word mark-top-level-syntax parse-definition
    define-syntax ;
```

(Factor is self hosted)
meta-circularity

Factor: *Language = Library*

Factor: *Language = Library*

- Control Flow

  Implemented as combinators

- Module System

  Implemented as syntax extensions

- Local (variables)

  Implemented as a library

- Object Orientation

  Implemented as syntax extensions

- Interactive Help System

  Implemented as a DSL in a library

STACK LANGUAGES ARE

PRETTY COOL

Questions?