# 61A Lecture 26

Monday, October 29

# Today's Topic: Handling Errors

# Today's Topic: Handling Errors

```
Sometimes, computers don't do exactly what we expect
```

# Today's Topic: Handling Errors

Sometimes, computers don't do exactly what we expect

- A function receives unexpected argument types

# Today's Topic: Handling Errors

Sometimes, computers don't do exactly what we expect

- A function receives unexpected argument types
- Some resource (such as a file) is not available

# Today's Topic: Handling Errors

Sometimes, computers don't do exactly what we expect

- A function receives unexpected argument types
- Some resource (such as a file) is not available
- A network connection is lost

# Today's Topic: Handling Errors

Sometimes, computers don't do exactly what we expect

- A function receives unexpected argument types
- Some resource (such as a file) is not available
- A network connection is lost


Grace Hopper's Notebook, 1947, Moth found in a Mark II Computer

# Exceptions

# Exceptions

A built-in mechanism in a programming language to declare and respond to exceptional conditions

# Exceptions

A built-in mechanism in a programming language to declare and respond to exceptional conditions

Python *raises* an exception whenever an error occurs

# Exceptions

A built-in mechanism in a programming language to declare and respond to exceptional conditions

Python *raises* an exception whenever an error occurs

Exceptions can be *handled* by the program, preventing a crash

# Exceptions

A built-in mechanism in a programming language to declare and respond to exceptional conditions

Python *raises* an exception whenever an error occurs

Exceptions can be *handled* by the program, preventing a crash

Unhandled exceptions will cause Python to halt execution

# Exceptions

A built-in mechanism in a programming language to declare and respond to exceptional conditions

Python *raises* an exception whenever an error occurs

Exceptions can be *handled* by the program, preventing a crash

Unhandled exceptions will cause Python to halt execution

**Mastering exceptions:**

# Exceptions

A built-in mechanism in a programming language to declare and respond to exceptional conditions

Python *raises* an exception whenever an error occurs

Exceptions can be *handled* by the program, preventing a crash

Unhandled exceptions will cause Python to halt execution

**Mastering exceptions:**

Exceptions are objects! They have classes with constructors.

# Exceptions

A built-in mechanism in a programming language to declare and respond to exceptional conditions

Python *raises* an exception whenever an error occurs

Exceptions can be *handled* by the program, preventing a crash

Unhandled exceptions will cause Python to halt execution

**Mastering exceptions:**

Exceptions are objects! They have classes with constructors.

They enable *non-local* continuations of control:

# Exceptions

A built-in mechanism in a programming language to declare and respond to exceptional conditions

Python *raises* an exception whenever an error occurs

Exceptions can be *handled* by the program, preventing a crash

Unhandled exceptions will cause Python to halt execution

**Mastering exceptions:**

Exceptions are objects! They have classes with constructors.

They enable *non-local* continuations of control:

If **f** calls **g** and **g** calls **h**, exceptions can shift control from **h** to **f** without waiting for **g** to return.

# Exceptions

A built-in mechanism in a programming language to declare and respond to exceptional conditions

Python *raises* an exception whenever an error occurs

Exceptions can be *handled* by the program, preventing a crash

Unhandled exceptions will cause Python to halt execution

**Mastering exceptions:**

Exceptions are objects! They have classes with constructors.

They enable *non-local* continuations of control:

If **f** calls **g** and **g** calls **h**, exceptions can shift control from **h** to **f** without waiting for **g** to return.

However, exception handling tends to be slow.

# Assert Statements

`Assert statements raise an exception of type AssertionError`

# Assert Statements

Assert statements raise an exception of type AssertionError

```
assert <expression>, <string>
```

# Assert Statements

Assert statements raise an exception of type AssertionError

```
assert <expression>, <string>
```

Assertions are designed to be used liberally and then disabled in "production" systems. "O" stands for optimized.

# Assert Statements

Assert statements raise an exception of type AssertionError

```
assert <expression>, <string>
```

Assertions are designed to be used liberally and then disabled in "production" systems. "O" stands for optimized.

```
python3 -O
```

# Assert Statements

Assert statements raise an exception of type AssertionError

```
assert <expression>, <string>
```

Assertions are designed to be used liberally and then disabled in "production" systems. "O" stands for optimized.

```
python3 -O
```

Whether assertions are enabled is governed by a bool `__debug__`

# Assert Statements

Assert statements raise an exception of type AssertionError

```
assert <expression>, <string>
```

Assertions are designed to be used liberally and then disabled in "production" systems. "O" stands for optimized.

```
python3 -O
```

Whether assertions are enabled is governed by a bool `__debug__`

Demo

# Raise Statements

# Raise Statements

Exceptions are raised with a raise statement.

# Raise Statements

Exceptions are raised with a raise statement.

```
raise <expression>
```

# Raise Statements

Exceptions are raised with a raise statement.

raise <expression>

<expression> must evaluate to an exception instance or class.

# Raise Statements

Exceptions are raised with a raise statement.

```
raise <expression>
```

<expression> must evaluate to an exception instance or class.

Exceptions are constructed like any other object; they are just instances of classes that inherit from BaseException.

## Raise Statements

Exceptions are raised with a raise statement.

```
raise <expression>
```

<expression> must evaluate to an exception instance or class.

Exceptions are constructed like any other object; they are just instances of classes that inherit from BaseException.

TypeError -- A function was passed the wrong number/type of argument

# Raise Statements

Exceptions are raised with a raise statement.

```
raise <expression>
```

<expression> must evaluate to an exception instance or class.

Exceptions are constructed like any other object; they are just instances of classes that inherit from BaseException.

TypeError -- A function was passed the wrong number/type of argument

NameError -- A name wasn't found

# Raise Statements

Exceptions are raised with a raise statement.

```
raise <expression>
```

<expression> must evaluate to an exception instance or class.

Exceptions are constructed like any other object; they are just instances of classes that inherit from BaseException.

TypeError -- A function was passed the wrong number/type of argument

NameError -- A name wasn't found

KeyError -- A key wasn't found in a dictionary

# Raise Statements

Exceptions are raised with a raise statement.

```
raise <expression>
```

<expression> must evaluate to an exception instance or class.

Exceptions are constructed like any other object; they are just instances of classes that inherit from BaseException.

TypeError -- A function was passed the wrong number/type of argument

NameError -- A name wasn't found

KeyError -- A key wasn't found in a dictionary

RuntimeError -- Catch-all for troubles during interpretation

# Try Statements

## Try Statements

Try statements handle exceptions

# Try Statements

Try statements handle exceptions

```
try:
    <try suite>
except <exception class> as <name>:
    <except suite>
...
```

# Try Statements

Try statements handle exceptions

```
try:
    <try suite>
except <exception class> as <name>:
    <except suite>
...
```

**Execution rule:**

# Try Statements

Try statements handle exceptions

```
try:
    <try suite>
except <exception class> as <name>:
    <except suite>
...
```

**Execution rule:**

The <try suite> is executed first;

# Try Statements

Try statements handle exceptions

```
try:
    <try suite>
except <exception class> as <name>:
    <except suite>
...
```

**Execution rule:**

The <try suite> is executed first;

If, during the course of executing the <try suite>,
an exception is raised that is not handled otherwise, and

# Try Statements

Try statements handle exceptions

```
try:
    <try suite>
except <exception class> as <name>:
    <except suite>
...
```

**Execution rule:**

The <try suite> is executed first;

If, during the course of executing the <try suite>,
an exception is raised that is not handled otherwise, and

If the class of the exception inherits from <exception class>, then

# Try Statements

Try statements handle exceptions

```
try:
    <try suite>
except <exception class> as <name>:
    <except suite>
...
```

**Execution rule:**

The <try suite> is executed first;

If, during the course of executing the <try suite>,
an exception is raised that is not handled otherwise, and

If the class of the exception inherits from <exception class>, then

The <except suite> is executed, with <name> bound to the exception

# Handling Exceptions

# Handling Exceptions

Exception handling can prevent a program from terminating

# Handling Exceptions

Exception handling can prevent a program from terminating

```
>>> try:
```

# Handling Exceptions

Exception handling can prevent a program from terminating

```
>>> try:
        x = 1/0
```

# Handling Exceptions

Exception handling can prevent a program from terminating

```
>>> try:
        x = 1/0
    except ZeroDivisionError as e:
```

# Handling Exceptions

Exception handling can prevent a program from terminating

```
>>> try:
        x = 1/0
    except ZeroDivisionError as e:
        print('handling a', type(e))
```

# Handling Exceptions

Exception handling can prevent a program from terminating

```
>>> try:
        x = 1/0
    except ZeroDivisionError as e:
        print('handling a', type(e))
        x = 0
```

# Handling Exceptions

Exception handling can prevent a program from terminating

```
>>> try:
        x = 1/0
    except ZeroDivisionError as e:
        print('handling a', type(e))
        x = 0

handling a <class 'ZeroDivisionError'>
```

# Handling Exceptions

Exception handling can prevent a program from terminating

```
>>> try:
        x = 1/0
    except ZeroDivisionError as e:
        print('handling a', type(e))
        x = 0

handling a <class 'ZeroDivisionError'>
>>> x
```

# Handling Exceptions

Exception handling can prevent a program from terminating

```
>>> try:
        x = 1/0
    except ZeroDivisionError as e:
        print('handling a', type(e))
        x = 0

handling a <class 'ZeroDivisionError'>
>>> x
0
```

# Handling Exceptions

Exception handling can prevent a program from terminating

```
>>> try:
        x = 1/0
    except ZeroDivisionError as e:
        print('handling a', type(e))
        x = 0

handling a <class 'ZeroDivisionError'>
>>> x
0
```

**Multiple try statements:** Control jumps to the except suite of the most recent try statement that handles that type of exception.

## Handling Exceptions

Exception handling can prevent a program from terminating

```
>>> try:
        x = 1/0
    except ZeroDivisionError as e:
        print('handling a', type(e))
        x = 0

handling a <class 'ZeroDivisionError'>
>>> x
0
```

**Multiple try statements:** Control jumps to the except suite of the most recent try statement that handles that type of exception.

Demo

# WWPD: What Would Python Do?

How will the Python interpreter respond?

# WWPD: What Would Python Do?

```
How will the Python interpreter respond?
```

# WWPD: What Would Python Do?

How will the Python interpreter respond?

```
def invert(x):
    result = 1/x  # Raises a ZeroDivisionError if x is 0
    print('Never printed if x is 0')
    return result

def invert_safe(x):
    try:
        return invert(x)
    except ZeroDivisionError as e:
        return str(e)
```

# WWPD: What Would Python Do?

How will the Python interpreter respond?

```
def invert(x):
    result = 1/x  # Raises a ZeroDivisionError if x is 0
    print('Never printed if x is 0')
    return result

def invert_safe(x):
    try:
        return invert(x)
    except ZeroDivisionError as e:
        return str(e)
```

```
>>> invert_safe(1/0)
```

# WWPD: What Would Python Do?

How will the Python interpreter respond?

```
def invert(x):
    result = 1/x  # Raises a ZeroDivisionError if x is 0
    print('Never printed if x is 0')
    return result

def invert_safe(x):
    try:
        return invert(x)
    except ZeroDivisionError as e:
        return str(e)
```

```
>>> invert_safe(1/0)

>>> try:
        invert_safe(0)
    except ZeroDivisionError as e:
        print('Handled!')
```

# WWPD: What Would Python Do?

How will the Python interpreter respond?

```
def invert(x):
    result = 1/x  # Raises a ZeroDivisionError if x is 0
    print('Never printed if x is 0')
    return result

def invert_safe(x):
    try:
        return invert(x)
    except ZeroDivisionError as e:
        return str(e)
```

```
>>> invert_safe(1/0)

>>> try:
        invert_safe(0)
    except ZeroDivisionError as e:
        print('Handled!')

>>> inverrrrt_safe(1/0)
```

# Reading Scheme Lists

# Reading Scheme Lists

A Scheme list is written as elements in parentheses:

# Reading Scheme Lists

A Scheme list is written as elements in parentheses:

```
(<element_0> <element_1> ... <element_n>)
```

# Reading Scheme Lists

A Scheme list is written as elements in parentheses:

```
(<element_0> <element_1> ... <element_n>)
```

A recursive Scheme list

# Reading Scheme Lists

A Scheme list is written as elements in parentheses:

```
(<element_0> <element_1> ... <element_n>)
```

A recursive Scheme list

# Reading Scheme Lists

A Scheme list is written as elements in parentheses:

(<element_0> <element_1> ... <element_n>)

A recursive Scheme list

# Reading Scheme Lists

A Scheme list is written as elements in parentheses:

```
(<element_0> <element_1> ... <element_n>)
```

A recursive Scheme list

Each <element> can be a combination or primitive.

# Reading Scheme Lists

A Scheme list is written as elements in parentheses:

```
(<element_0> <element_1> ... <element_n>)
```

A recursive Scheme list

Each <element> can be a combination or primitive.

```
(+ (* 3 (+ (* 2 4) (+ 3 5))) (+ (- 10 7) 6))
```

# Reading Scheme Lists

A Scheme list is written as elements in parentheses:

(<element_0> <element_1> ... <element_n>)

A recursive Scheme list

Each <element> can be a combination or primitive.

```
(+ (* 3 (+ (* 2 4) (+ 3 5))) (+ (- 10 7) 6))
```

The task of *parsing* a language involves coercing a string representation of an expression to the expression itself.

# Reading Scheme Lists

A Scheme list is written as elements in parentheses:

```
(<element_0> <element_1> ... <element_n>)
```

A recursive Scheme list

Each <element> can be a combination or primitive.

```
(+ (* 3 (+ (* 2 4) (+ 3 5))) (+ (- 10 7) 6))
```

The task of *parsing* a language involves coercing a string representation of an expression to the expression itself.

Parsers must validate that expressions are well-formed.

# Reading Scheme Lists

A Scheme list is written as elements in parentheses:

(<element_0> <element_1> ... <element_n>)

A recursive Scheme list

Each <element> can be a combination or primitive.

```
(+ (* 3 (+ (* 2 4) (+ 3 5))) (+ (- 10 7) 6))
```

The task of *parsing* a language involves coercing a string representation of an expression to the expression itself.

Parsers must validate that expressions are well-formed.

Demo (http://inst.eecs.berkeley.edu/~cs61a/fa12/projects/scalc/scheme_reader.py.html)

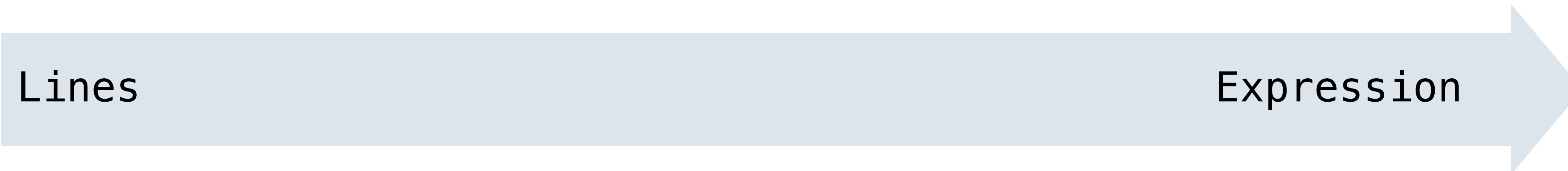
# Parsing

# Parsing

A Parser takes a sequence of lines and returns an expression.

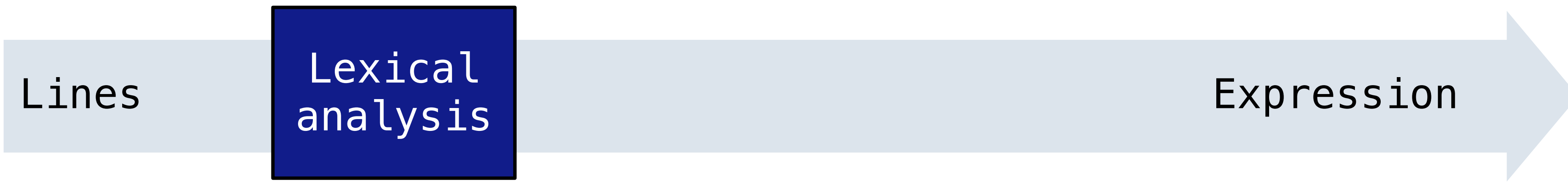
# Parsing

A Parser takes a sequence of lines and returns an expression.



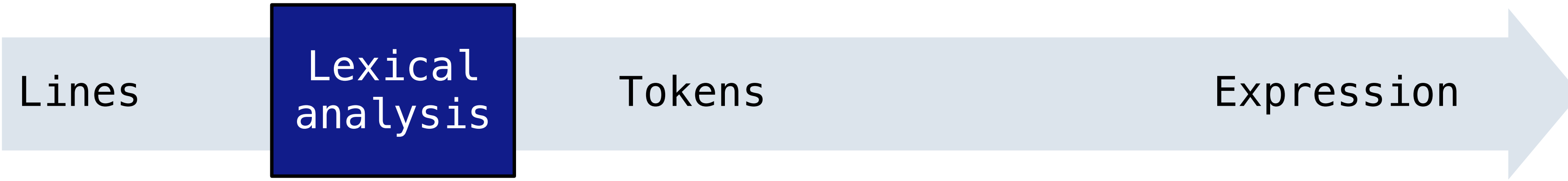
# Parsing

A Parser takes a sequence of lines and returns an expression.



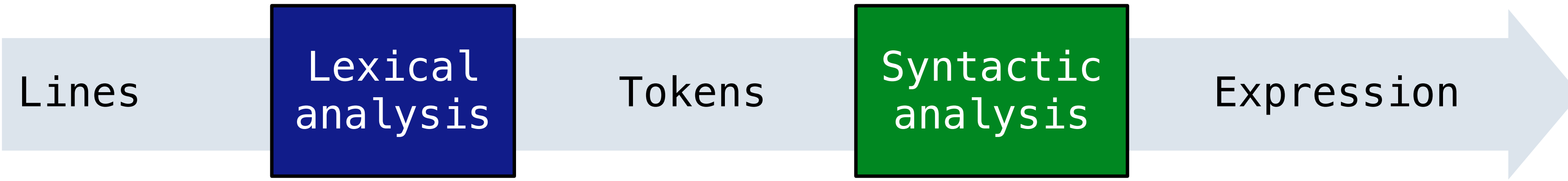
# Parsing

A Parser takes a sequence of lines and returns an expression.



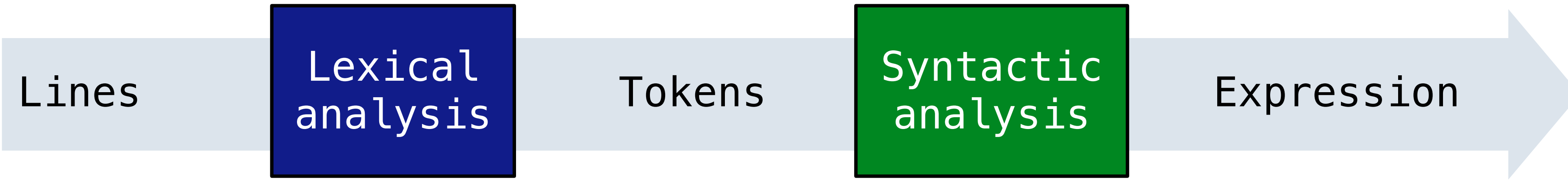
# Parsing

A Parser takes a sequence of lines and returns an expression.



# Parsing

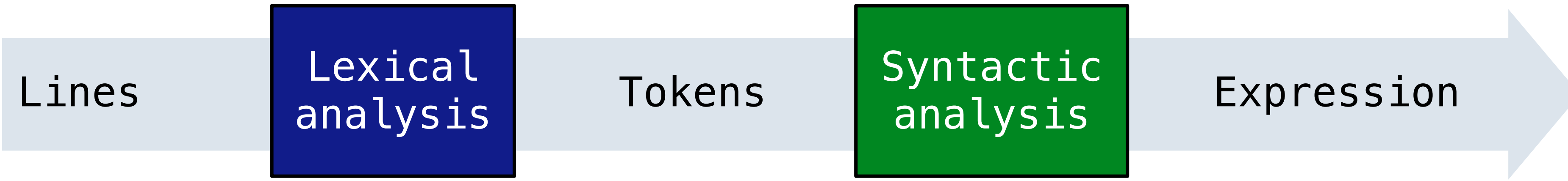
A Parser takes a sequence of lines and returns an expression.




```
['(+ 1',
 '   (- 23)',
 '   (* 4 5.6))']
```

# Parsing

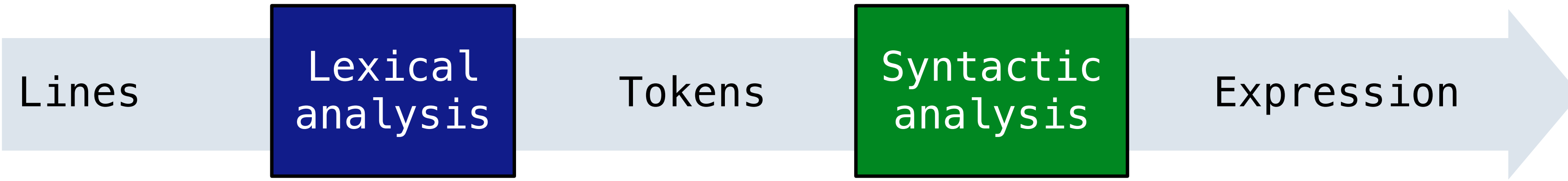
A Parser takes a sequence of lines and returns an expression.




```
['(+ 1',
 '   (- 23)',
 '   (* 4 5.6))']
```

# Parsing

A Parser takes a sequence of lines and returns an expression.

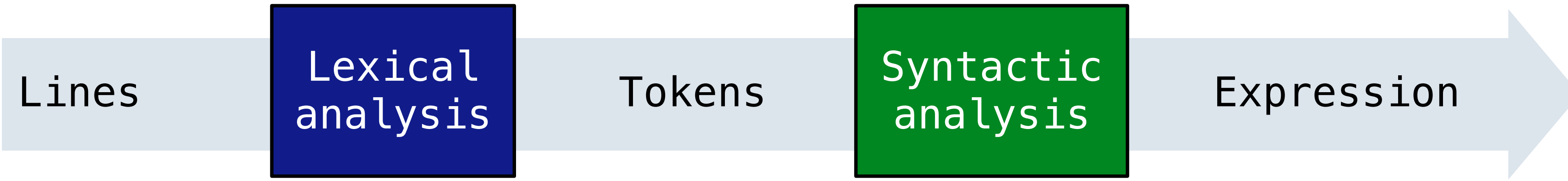



```
['(+ 1',
 '   (- 23)',
 '   (* 4 5.6))']
```

```
['(', '+', 1]
```

# Parsing

A Parser takes a sequence of lines and returns an expression.

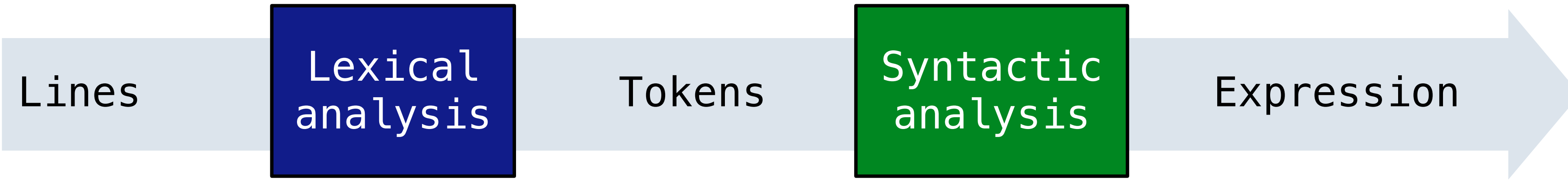



```
['(+ 1',
 '   (- 23)',
 '   (* 4 5.6))']
```

```
['(', '+', 1]
['(', '-', 23, ')']
```

# Parsing

A Parser takes a sequence of lines and returns an expression.

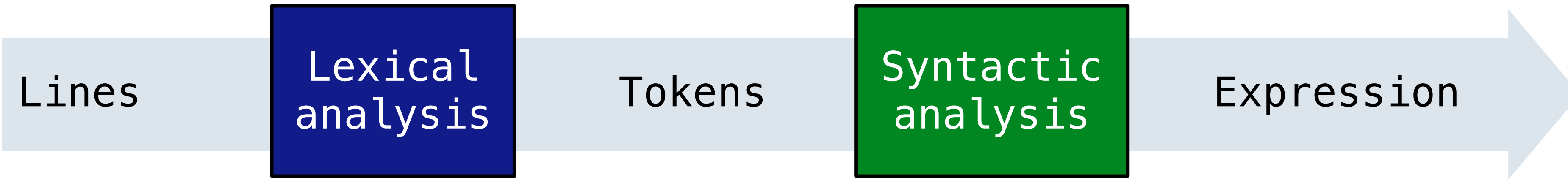



```
['(+ 1',
 '   (- 23)',
 '   (* 4 5.6))']
```

```
['(', '+', 1]
['(', '-', 23, ')']
```

# Parsing

A Parser takes a sequence of lines and returns an expression.

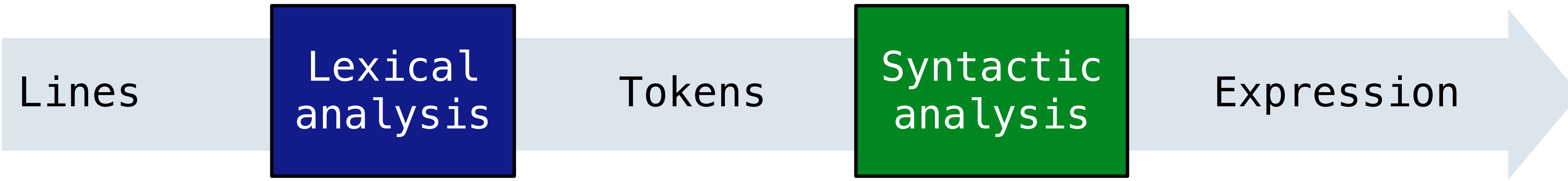



```
['(+ 1',
 '   (- 23)',
 '   (* 4 5.6))']
```

```
['(', '+', 1]
['(', '-', 23, ')']
```

# Parsing

A Parser takes a sequence of lines and returns an expression.

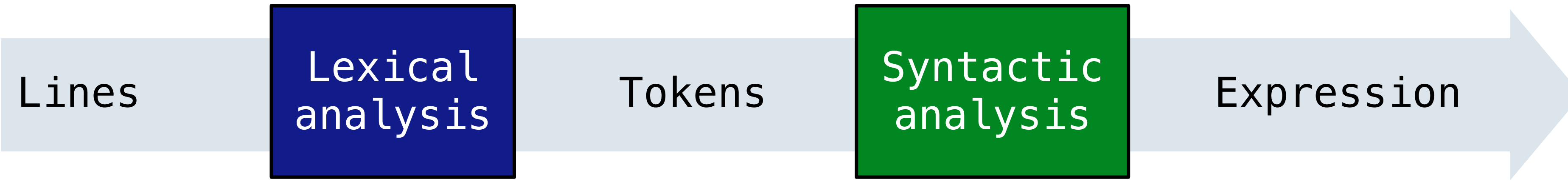



```
['(+ 1',
 '   (- 23)',
 '   (* 4 5.6))']
```

```
['(', '+', 1]
['(', '-', 23, ')']
['(', '*', 4, 5.6, ')', ')']
```

# Parsing

A Parser takes a sequence of lines and returns an expression.




```
['(+ 1',
 '   (- 23)',
 '   (* 4 5.6))']
```

```
['(', '+', 1]
['(', '-', 23, ')']
['(', '*', 4, 5.6, ')', ')']
```

# Parsing

A Parser takes a sequence of lines and returns an expression.

Lines → **Lexical analysis** → Tokens → **Syntactic analysis** → Expression

```
['(+ 1',                 ['(', '+', 1]
 '   (- 23)',            ['(', '-', 23, ')']
 '   (* 4 5.6))']        ['(', '*', 4, 5.6, ')', ')']
```

- Iterative process

# Parsing

A Parser takes a sequence of lines and returns an expression.

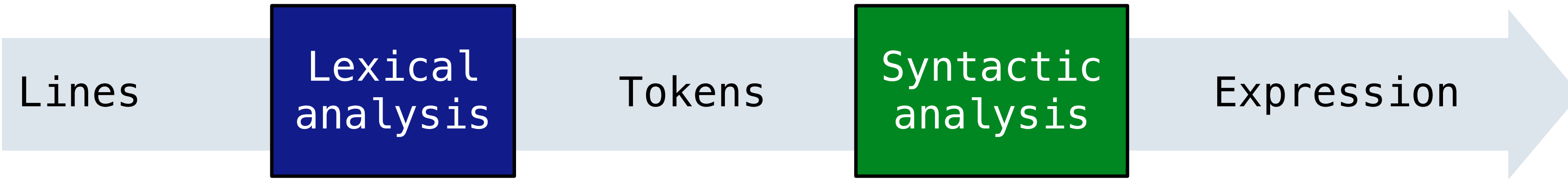
Lines → **Lexical analysis** → Tokens → **Syntactic analysis** → Expression

```
['(+ 1',              ['(', '+', 1]
 '   (- 23)',         ['(', '-', 23, ')']
 '   (* 4 5.6))']     ['(', '*', 4, 5.6, ')', ')']
```

- Iterative process
- Checks for malformed tokens

# Parsing

A Parser takes a sequence of lines and returns an expression.



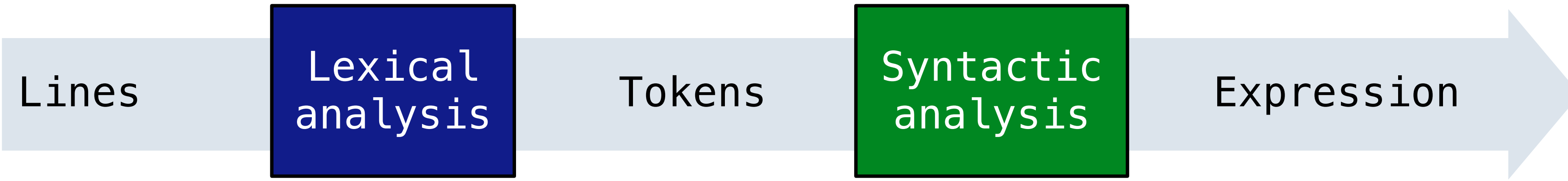

```
['(+ 1',
 '   (- 23)',
 '   (* 4 5.6))']
```

```
['(', '+', 1]
['(', '-', 23, ')']
['(', '*', 4, 5.6, ')', ')']
```

- Iterative process
- Checks for malformed tokens
- Determines types of tokens

# Parsing

A Parser takes a sequence of lines and returns an expression.




```
['(+ 1',
 '   (- 23)',
 '   (* 4 5.6))']
```

```
['(', '+', 1]
['(', '-', 23, ')']
['(', '*', 4, 5.6, ')', ')']
```

- Iterative process
- Checks for malformed tokens
- Determines types of tokens
- Processes one line at a time

# Parsing

A Parser takes a sequence of lines and returns an expression.

Lines → **Lexical analysis** → Tokens → **Syntactic analysis** → Expression

```
['(+ 1',                  ['(', '+', 1]
  '   (- 23)',            ['(', '-', 23, ')']
  '   (* 4 5.6))']        ['(', '*', 4, 5.6, ')', ')']
```

- Iterative process
- Checks for malformed tokens
- Determines types of tokens
- Processes one line at a time

# Parsing

A Parser takes a sequence of lines and returns an expression.

Lines → **Lexical analysis** → Tokens → **Syntactic analysis** → Expression

```
['(+ 1',
 '   (- 23)',
 '   (* 4 5.6))']
```

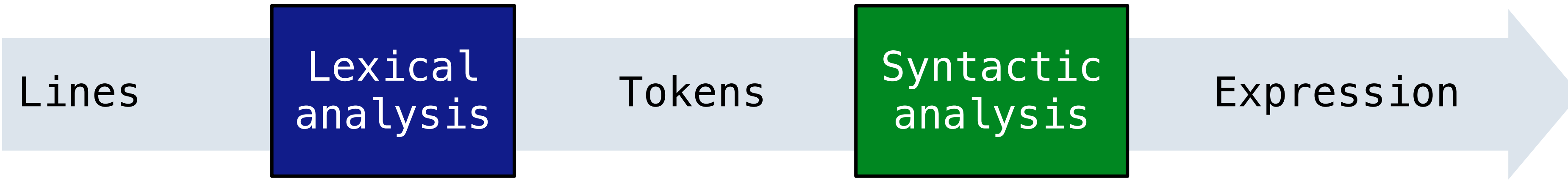
```
['(', '+', 1]
['(', '-', 23, ')']
['(', '*', 4, 5.6, ')', ')']
```

```
Pair('+', Pair(1, ...))
```

- Iterative process
- Checks for malformed tokens
- Determines types of tokens
- Processes one line at a time

# Parsing

A Parser takes a sequence of lines and returns an expression.




```
['(+ 1',
  '   (- 23)',
  '   (* 4 5.6))']
```

```
['(', '+', 1]
['(', '-', 23, ')']
['(', '*', 4, 5.6, ')', ')']
```

```
Pair('+', Pair(1, ...))
```

*printed as*

```
(+ 1 (- 23) (* 4 5.6))
```

- Iterative process
- Checks for malformed tokens
- Determines types of tokens
- Processes one line at a time

# Parsing

A Parser takes a sequence of lines and returns an expression.

Lines → **Lexical analysis** → Tokens → **Syntactic analysis** → Expression

```
['(+ 1',
  '   (- 23)',
  '   (* 4 5.6))']
```

```
['(', '+', 1]
['(', '-', 23, ')']
['(', '*', 4, 5.6, ')', ')']
```

```
Pair('+', Pair(1, ...))
```

*printed as*

```
(+ 1 (- 23) (* 4 5.6))
```

- Iterative process
- Checks for malformed tokens
- Determines types of tokens
- Processes one line at a time

- Tree-recursive process

# Parsing

A Parser takes a sequence of lines and returns an expression.

Lines → **Lexical analysis** → Tokens → **Syntactic analysis** → Expression

```
['(+ 1',
 '   (- 23)',
 '   (* 4 5.6))']
```

```
['(', '+', 1]
['(', '-', 23, ')']
['(', '*', 4, 5.6, ')', ')']
```

```
Pair('+', Pair(1, ...))
```

*printed as*

```
(+ 1 (- 23) (* 4 5.6))
```

- Iterative process
- Checks for malformed tokens
- Determines types of tokens
- Processes one line at a time

- Tree-recursive process
- Balances parentheses

# Parsing

A Parser takes a sequence of lines and returns an expression.

Lines → **Lexical analysis** → Tokens → **Syntactic analysis** → Expression

```
['(+ 1',
  '   (- 23)',
  '   (* 4 5.6))']
```

```
['(', '+', 1]
['(', '-', 23, ')']
['(', '*', 4, 5.6, ')', ')']
```

```
Pair('+', Pair(1, ...))
```

*printed as*

```
(+ 1 (- 23) (* 4 5.6))
```

- Iterative process
- Checks for malformed tokens
- Determines types of tokens
- Processes one line at a time

- Tree-recursive process
- Balances parentheses
- Returns tree structure

# Parsing

A Parser takes a sequence of lines and returns an expression.

Lines → **Lexical analysis** → Tokens → **Syntactic analysis** → Expression

```
['(+ 1',
  '   (- 23)',
  '   (* 4 5.6))']
```

```
['(', '+', 1]
['(', '-', 23, ')']
['(', '*', 4, 5.6, ')', ')']
```

```
Pair('+', Pair(1, ...))
```

*printed as*

```
(+ 1 (- 23) (* 4 5.6))
```

- Iterative process
- Checks for malformed tokens
- Determines types of tokens
- Processes one line at a time

- Tree-recursive process
- Balances parentheses
- Returns tree structure
- Processes multiple lines

# Recursive Syntactic Analysis

## Recursive Syntactic Analysis

A predictive recursive descent parser inspects only *k* tokens to decide how to proceed, for some fixed *k*.

# Recursive Syntactic Analysis

A predictive recursive descent parser inspects only *k* tokens to decide how to proceed, for some fixed *k*.

*Can English be parsed via predictive recursive descent?*

# Recursive Syntactic Analysis

A predictive recursive descent parser inspects only *k* tokens to decide how to proceed, for some fixed *k*.

*Can English be parsed via predictive recursive descent?*

The horse raced past the barn fell.

# Recursive Syntactic Analysis

A predictive recursive descent parser inspects only *k* tokens to decide how to proceed, for some fixed *k*.

*Can English be parsed via predictive recursive descent?*

The horse ~~raced~~ past the barn fell.
ridden

# Recursive Syntactic Analysis

A predictive recursive descent parser inspects only *k* tokens to decide how to proceed, for some fixed *k*.

*Can English be parsed via predictive recursive descent?*

The horse ~~raced~~ past the barn fell.

ridden

(that was)

# Recursive Syntactic Analysis

A predictive recursive descent parser inspects only *k* tokens to decide how to proceed, for some fixed *k*.

*Can English be parsed via predictive recursive descent?*

sentence subject

The horse ~~raced~~ past the barn fell.

ridden

(that was)

# Recursive Syntactic Analysis

A predictive recursive descent parser inspects only *k* tokens to decide how to proceed, for some fixed *k*.

*Can English be parsed via predictive recursive descent?*

sentence subject

The horse ~~raced~~ past the barn fell.

ridden

*(that was)*

You got
Gardenpath'd!

# Syntactic Analysis

# Syntactic Analysis

Syntactic analysis identifies the hierarchical structure of an expression, which may be nested.

# Syntactic Analysis

Syntactic analysis identifies the hierarchical structure of an expression, which may be nested.

Each call to scheme_read consumes the input tokens for exactly one expression.

# Syntactic Analysis

Syntactic analysis identifies the hierarchical structure of an expression, which may be nested.

Each call to scheme_read consumes the input tokens for exactly one expression.

```
'(', '+', 1, '(', '-', 23, ')', '(', '*', 4, 5.6, ')', ')'
```

# Syntactic Analysis

Syntactic analysis identifies the hierarchical structure of an expression, which may be nested.

Each call to scheme_read consumes the input tokens for exactly one expression.

```
'(', '+', 1, '(', '-', 23, ')', '(', '*', 4, 5.6, ')', ')'
```

# Syntactic Analysis

Syntactic analysis identifies the hierarchical structure of an expression, which may be nested.

Each call to scheme_read consumes the input tokens for exactly one expression.

```
'(', '+', 1, '(', '-', 23, ')', '(', '*', 4, 5.6, ')', ')'
```

**Recursive call:** scheme_read sub-expressions and combine them

# Syntactic Analysis

Syntactic analysis identifies the hierarchical structure of an expression, which may be nested.

Each call to scheme_read consumes the input tokens for exactly one expression.

```
'(', '+', 1, '(', '-', 23, ')', '(', '*', 4, 5.6, ')', ')'
```

**Recursive call:** scheme_read sub-expressions and combine them

# Syntactic Analysis

Syntactic analysis identifies the hierarchical structure of an expression, which may be nested.

Each call to scheme_read consumes the input tokens for exactly one expression.

```
'(', '+', 1, '(', '-', 23, ')', '(', '*', 4, 5.6, ')', ')'
```

**Recursive call:** scheme_read sub-expressions and combine them

**Base case:** symbols and numbers

# Syntactic Analysis

Syntactic analysis identifies the hierarchical structure of an expression, which may be nested.

Each call to scheme_read consumes the input tokens for exactly one expression.

```
'(', '+', 1, '(', '-', 23, ')', '(', '*', 4, 5.6, ')', ')'
```

**Recursive call:** scheme_read sub-expressions and combine them

**Base case:** symbols and numbers

# Syntactic Analysis

Syntactic analysis identifies the hierarchical structure of an expression, which may be nested.

Each call to scheme_read consumes the input tokens for exactly one expression.

```
'(', '+', 1, '(', '-', 23, ')', '(', '*', 4, 5.6, ')', ')'
```

**Recursive call:** scheme_read sub-expressions and combine them

**Base case:** symbols and numbers

# Syntactic Analysis

Syntactic analysis identifies the hierarchical structure of an expression, which may be nested.

Each call to scheme_read consumes the input tokens for exactly one expression.

```
'(', '+', 1, '(', '-', 23, ')', '(', '*', 4, 5.6, ')', ')'
```

**Recursive call:** scheme_read sub-expressions and combine them

**Base case:** symbols and numbers

# Syntactic Analysis

Syntactic analysis identifies the hierarchical structure of an expression, which may be nested.

Each call to scheme_read consumes the input tokens for exactly one expression.

```
'(', '+', 1, '(', '-', 23, ')', '(', '*', 4, 5.6, ')', ')'
```

**Recursive call:** scheme_read sub-expressions and combine them

**Base case:** symbols and numbers

# Syntactic Analysis

Syntactic analysis identifies the hierarchical structure of an expression, which may be nested.

Each call to scheme_read consumes the input tokens for exactly one expression.

```
'(', '+', 1, '(', '-', 23, ')', '(', '*', 4, 5.6, ')', ')'
```

**Recursive call:** scheme_read sub-expressions and combine them

**Base case:** symbols and numbers

# Syntactic Analysis

Syntactic analysis identifies the hierarchical structure of an expression, which may be nested.

Each call to scheme_read consumes the input tokens for exactly one expression.

```
'(', '+', 1, '(', '-', 23, ')', '(', '*', 4, 5.6, ')', ')'
```

**Recursive call:** scheme_read sub-expressions and combine them

**Base case:** symbols and numbers

Demo (http://inst.eecs.berkeley.edu/~cs61a/fa12/projects/scalc/scheme_reader.py.html)