Announcements

- Homework 7 due Tuesday 11/5 @ 11:59pm.
Announcements

• Homework 7 due Tuesday 11/5 @ 11:59pm.
• Project 1 composition revisions due Thursday 11/7 @ 11:59pm.
Heard on the Dread Pirate Lambda's Fibbonautical Voyage
Heard on the Dread Pirate Lambda's Fibbonautical Voyage
Heard on the Dread Pirate Lambda's Fibbonautical Voyage

What do people fear most about the Dread Pirate Lambda?
Heard on the Dread Pirate Lambda's Fibbonautical Voyage

What do people fear most about the Dread Pirate Lambda?

His eval ways!
Heard on the Dread Pirate Lambda's Fibbonautical Voyage

What do people fear most about the Dread Pirate Lambda?

His eval ways!

When does the Dread Pirate Lambda finally stop plundering?
Heard on the Dread Pirate Lambda's Fibbonautical Voyage

What do people fear most about the Dread Pirate Lambda?

His eval ways!

When does the Dread Pirate Lambda finally stop plundering?

The base case!
Heard on the Dread Pirate Lambda's Fibonacciatical Voyage

What do people fear most about the Dread Pirate Lambda?

    His eval ways!

When does the Dread Pirate Lambda finally stop plundering?

    The base case!

What did the DPL say when he dropped his fruit overboard?
Heard on the Dread Pirate Lambda's Fibbonautical Voyage

What do people fear most about the Dread Pirate Lambda?

His eval ways!

When does the Dread Pirate Lambda finally stop plundering?

The base case!

What did the DPL say when he dropped his fruit overboard?

(Oh no, I've lost my pear in the seas!)
Exceptions
Today's Topic: Handling Errors
Today's Topic: Handling Errors

Sometimes, computer programs behave in non-standard ways
Today's Topic: Handling Errors

Sometimes, computer programs behave in non-standard ways

- A function receives an argument value of an improper type
Today's Topic: Handling Errors

Sometimes, computer programs behave in non-standard ways

• A function receives an argument value of an improper type
• Some resource (such as a file) is not available
Today's Topic: Handling Errors

Sometimes, computer programs behave in non-standard ways:

• A function receives an argument value of an improper type
• Some resource (such as a file) is not available
• A network connection is lost in the middle of data transmission
Today's Topic: Handling Errors

Sometimes, computer programs behave in non-standard ways

- A function receives an argument value of an improper type
- Some resource (such as a file) is not available
- A network connection is lost in the middle of data transmission

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 the interpreter from halting.
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 the interpreter from halting.

Unhandled exceptions will cause Python to halt execution and print a stack trace.
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 the interpreter from halting.

Unhandled exceptions will cause Python to halt execution and print a *stack trace*.

**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 the interpreter from halting.

Unhandled exceptions will cause Python to halt execution and print a stack trace.

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 the interpreter from halting.

Unhandled exceptions will cause Python to halt execution and print a *stack trace*.

**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 the interpreter from halting.

Unhandled exceptions will cause Python to halt execution and print a stack trace.

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 the interpreter from halting.

Unhandled exceptions will cause Python to halt execution and print a stack trace.

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.

(Exception handling tends to be slow.)
Raising Exceptions
Assert Statements

Assert statements raise an exception of type AssertionError
Assert Statements

Assert statements raise an exception of type AssertionError

```python
assert <expression>, <string>
```
Assert Statements

Assert statements raise an exception of type AssertionError

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

Assertions are designed to be used liberally. They can be ignored to increase efficiency by running Python with the "-O" flag. "O" stands for optimized.
Assert Statements

Assert statements raise an exception of type AssertionError

assert <expression>, <string>

Assertions are designed to be used liberally. They can be ignored to increase efficiency by running Python with the "-O" flag. "0" stands for optimized.

python3 -O
Assert Statements

Assert statements raise an exception of type AssertionError

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

Assertions are designed to be used liberally. They can be ignored to increase efficiency by running Python with the "-O" flag. "0" stands for optimized.

```shell
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. They can be ignored to increase efficiency by running Python with the "-O" flag. "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.

```python
raise <expression>
```
Exceptions are raised with a raise statement.

```
raise <expression>
```

<expression> must evaluate to a subclass of BaseException or an instance of one.
Raise Statements

Exceptions are raised with a raise statement.

```
raise <expression>
```

<expression> must evaluate to a subclass of BaseException or an instance of one.

Exceptions are constructed like any other object. E.g., TypeError('Bad argument!')
Raise Statements

Exceptions are raised with a raise statement.

```
raise <expression>
```

<expression> must evaluate to a subclass of BaseException or an instance of one.

Exceptions are constructed like any other object. E.g., TypeError('Bad argument!')

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

Exceptions are raised with a raise statement.

```python
raise <expression>
```

<expression> must evaluate to a subclass of BaseException or an instance of one.

Exceptions are constructed like any other object. E.g., `TypeError('Bad argument!')`

- **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 a subclass of BaseException or an instance of one.

Exceptions are constructed like any other object. E.g., `TypeError('Bad argument!')`

- **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.

```python
raise <expression>
```

<expression> must evaluate to a subclass of BaseException or an instance of one.

Exceptions are constructed like any other object. E.g., `TypeError('Bad argument!')`

- **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
Raise Statements

Exceptions are raised with a raise statement.

```
raise <expression>
```

<expression> must evaluate to a subclass of BaseException or an instance of one.

Exceptions are constructed like any other object. E.g., `TypeError('Bad argument!')`

- **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

(Demo)
Try Statements
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

```python
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

```python
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

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

```python
>>> try:
    x = 1/0
```
Handling Exceptions

Exception handling can prevent a program from terminating

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

```python
```
Handling Exceptions

Exception handling can prevent a program from terminating

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

Exception handling can prevent a program from terminating

```python
>>> 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

```python
>>> 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

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

handling a <class 'ZeroDivisionError'>
```

>>> x
12
Handling Exceptions

Exception handling can prevent a program from terminating

```python
>>> 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

```python
>>> 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

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

```python
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?

```python
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)
```

```text
ZeroDivisionError: division by zero
```
**WWPD: What Would Python Do?**

How will the Python interpreter respond?

```python
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:
```
**WWPD: What Would Python Do?**

How will the Python interpreter respond?

```python
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)
```
WWPD: What Would Python Do?

How will the Python interpreter respond?

```python
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:
```

**WWPD: What Would Python Do?**

How will the Python interpreter respond?

```python
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?

```python
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)
```
Interpreters
Reading Scheme Lists
Reading Scheme Lists

A Scheme list is written as elements in parentheses:
A Scheme list is written as elements in parentheses:

\[ (<\text{element}_0> \ <\text{element}_1> \ ... \ <\text{element}_n>) \]
Reading Scheme Lists

A Scheme list is written as elements in parentheses:

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

Each <element> can be a combination or primitive.
A Scheme list is written as elements in parentheses:

\(<\text{element}_0> \ <\text{element}_1> \ ... \ <\text{element}_n>\)

Each \(<\text{element}>\) 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:

\[(\text{element} _0 \text{ element} _1 \ldots \text{element} _n)\]

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:

\[(\langle\text{element}_0\rangle \ \langle\text{element}_1\rangle \ \ldots \ \langle\text{element}_n\rangle)\]

Each \langle\text{element}\rangle can be a combination or primitive.

\[(+ \ (\ast \ 3 \ (+ \ (\ast \ 2 \ 4) \ (+ \ 3 \ 5))) \ (+ \ (- \ 10 \ 7) \ 6))\]

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>)

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)
A Scheme list is written as elements in parentheses:

\[(\text{<element}_0\ \text{<element}_1\ \ldots\ \text{<element}_n})\]

Each \text{<element>} can be a combination or primitive.

\[(+\ (*\ 3\ (+\ (*\ 2\ 4)\ (+\ 3\ 5)))\ (+\ (-\ 10\ 7)\ 6))\]

The task of \textit{parsing} a language involves coercing a string representation of an expression to the expression itself.

Parsers must validate that expressions are well-formed.

(Demo)

A Scheme list is written as elements in parentheses:

\[(\text{element}_0 \ \text{element}_1 \ \ldots \ \text{element}_n)\]

Each \text{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)

Reading Scheme Lists

A Scheme list is written as elements in parentheses:

\[(\text{element}_0 \ \text{element}_1 \ \ldots \ \text{element}_n)\]

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)

Parsing
Parsing
Parsing

A Parser takes text and returns an expression.
A Parser takes text and returns an expression.
A Parser takes text and returns an expression.
A Parser takes text and returns an expression.
A Parser takes text and returns an expression.
A Parser takes text and returns an expression.

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

A Parser takes text and returns an expression.

```
(+ 1
'   (- 23)
'   (* 4 5.6))
```
A Parser takes text and returns an expression.

Text: '(+ 1
   (- 23)
   (* 4 5.6))'

Lexical analysis

Tokens: '(', '+', 1

Syntactic analysis

Expression
A Parser takes text and returns an expression.
Parsing

A Parser takes text and returns an expression.

Text: 

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

Lexical analysis:

Tokens:

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

Syntactic analysis: 

Expression:
A Parser takes text and returns an expression.
A Parser takes text and returns an expression.

Text

Lexical analysis

Tokens

Syntactic analysis

Expression

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

'(', '+', 1
'(', '-', 23, ')
'(', '*', 4, 5.6, ')
')'
A Parser takes text and returns an expression.
A Parser takes text and returns an expression.

- Iterative process

Text → Lexical analysis → Tokens → Syntactic analysis → Expression

• Parsing

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

• Iterative process
A Parser takes text and returns an expression.

- Iterative process
- Checks for malformed tokens
A Parser takes text and returns an expression.

- **Lexical analysis**
  - Iterative process
  - Checks for malformed tokens
  - Determines types of tokens

- **Tokens**

- **Syntactic analysis**

- **Expression**

Text: `'(+ 1' '(- 23)' '(* 4 5.6))'`

Expression: `'(' ' + ' 1 '(', ' - ', 23, ')', '(', ' * ', 4, 5.6, ')', ')')'
A Parser takes text and returns an expression.

- Iterative process
- Checks for malformed tokens
- Determines types of tokens
- Processes one line at a time
A Parser takes text and returns an expression.

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

A Parser takes text and returns an expression.

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

- **Tokens**

- **Syntactic analysis**

- **Expression**

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

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

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

A Parser takes text and returns an expression.

- Iterative process
- Checks for malformed tokens
- Determines types of tokens
- Processes one line at a time
A Parser takes text and returns an expression.

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

- **Syntactic analysis**
  - Tree-recursive process

<table>
<thead>
<tr>
<th>Text</th>
<th>Tokens</th>
<th>Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>'(+ 1'</td>
<td>'(, '+', 1</td>
<td>Pair('+', Pair(1, ...))</td>
</tr>
<tr>
<td>' ( - 23)'</td>
<td>'(, '-', 23, ')'</td>
<td>printed as</td>
</tr>
<tr>
<td>' ) (* 4 5.6))'</td>
<td>'(, '*', 4, 5.6, ')', ')'</td>
<td>(+ 1 (− 23) (* 4 5.6))</td>
</tr>
</tbody>
</table>
Parsing

A Parser takes text and returns an expression.

Text

Lexical analysis

Tokens

Syntactic analysis

Expression

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

• Tree-recursive process
• Balances parentheses

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

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

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

printed as

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

A Parser takes text and returns an expression.

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

- **Syntactic analysis**
  - Tree-recursive process
  - Balances parentheses
  - Returns tree structure

Text: 

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

Expression: 

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

Printed as:

```
(+ 1 (- 23) (* 4 5.6))
```
A Parser takes text and returns an expression.

- **Lexical analysis**
  - Tokens
  - Text
  - Tokens
  - Expression

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

- **Expression**
  - 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?

The horse-raced past the barn fell. (that was)
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.

**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.

**Base case:** symbols and numbers

**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, ')', ')
```

**Base case:** symbols and numbers

**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, ')', ')
```

**Base case:** symbols and numbers

**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, ')', ')
```

**Base case:** symbols and numbers

**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), ')
```

**Base case:** symbols and numbers

**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, ')', ')
```

**Base case:** symbols and numbers

**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, ')', ')

Base case: symbols and numbers

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, ')', ')

Base case: symbols and numbers

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, ')', ')
```

**Base case:** symbols and numbers

**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, ')', ')
```

**Base case**: symbols and numbers

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

(Demo)