61A Lecture 29

Monday, November 7

Homework: Huffman Encoding Trees

Homework: Huffman Encoding Trees

Efficient encoding of strings as ones and zeros (bits).

Homework: Huffman Encoding Trees

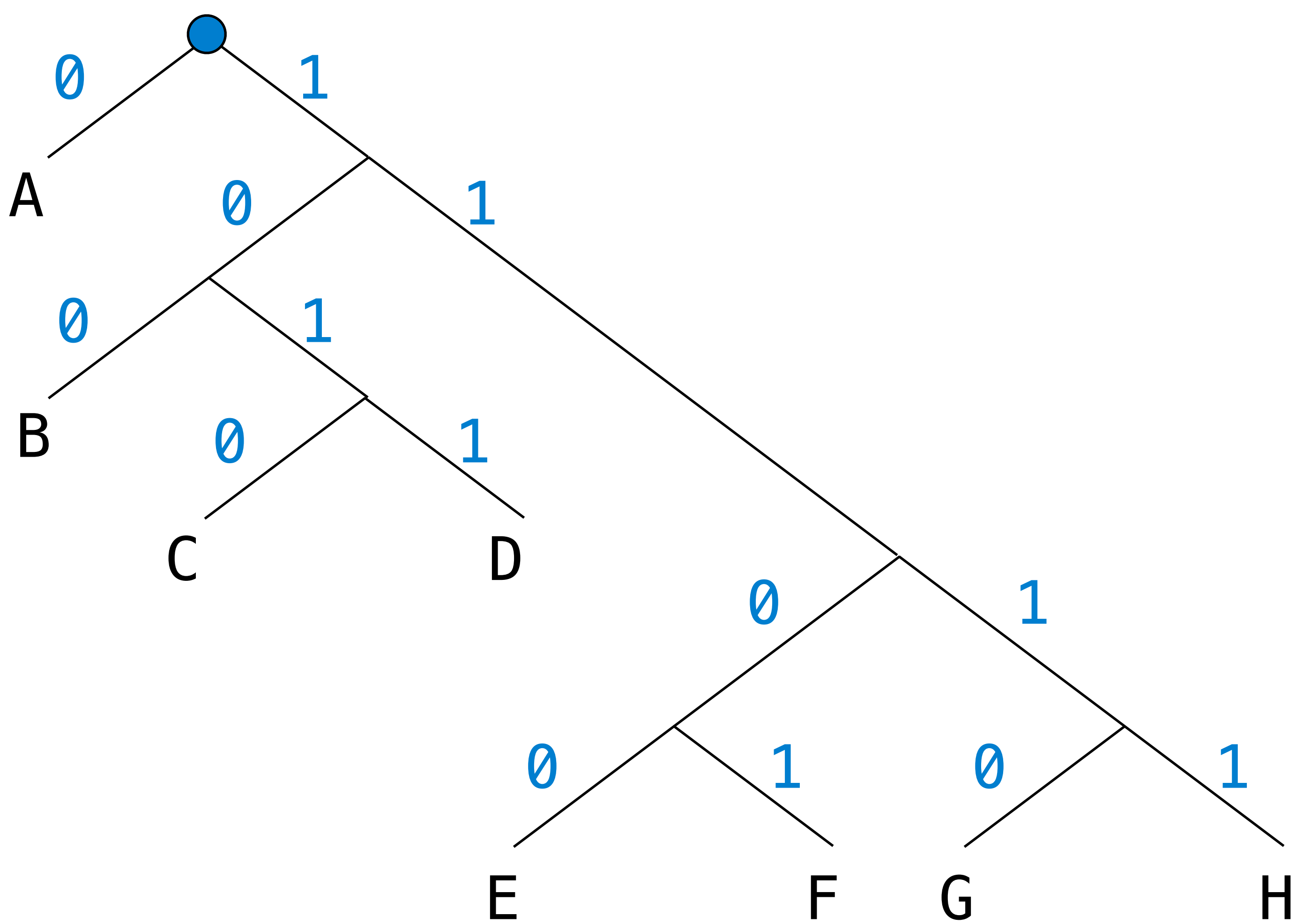
Efficient encoding of strings as ones and zeros (bits).

A 0	C 1010	E 1100	G 1110
B 100	D 1011	F 1101	H 1111

Homework: Huffman Encoding Trees

Efficient encoding of strings as ones and zeros (bits).

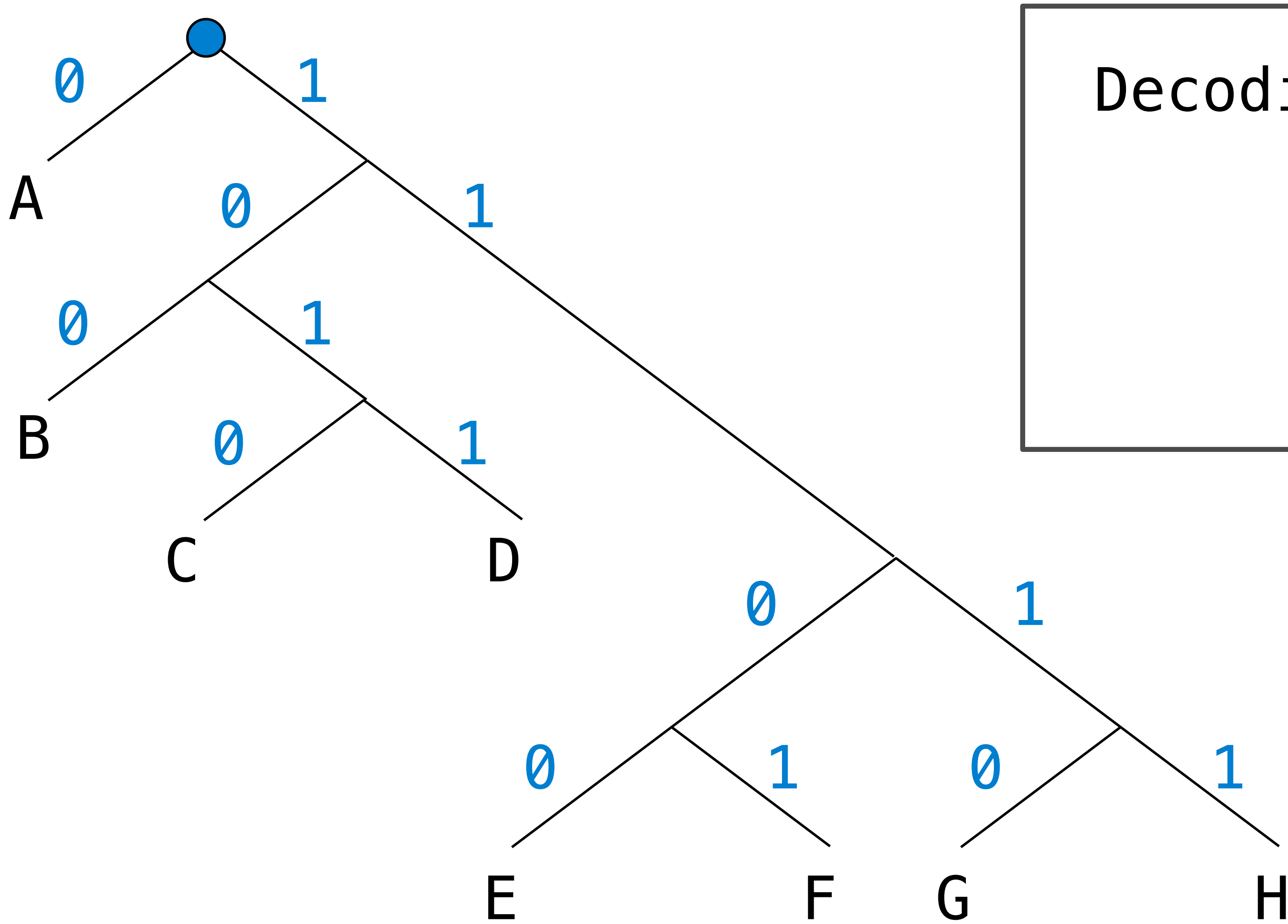
A 0	C 1010	E 1100	G 1110
B 100	D 1011	F 1101	H 1111



Homework: Huffman Encoding Trees

Efficient encoding of strings as ones and zeros (bits).

A 0	C 1010	E 1100	G 1110
B 100	D 1011	F 1101	H 1111

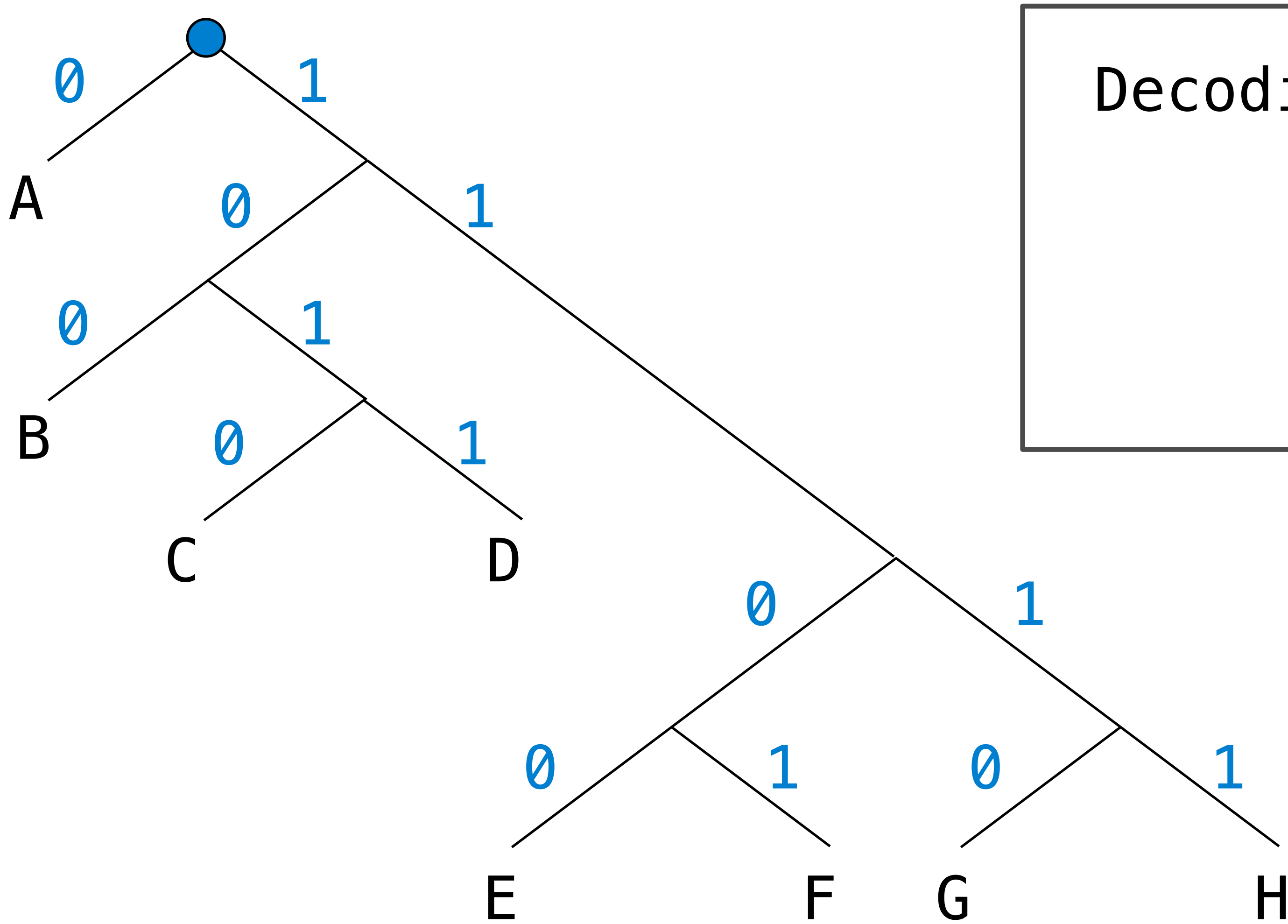


Decoding a sequence of bits:

Homework: Huffman Encoding Trees

Efficient encoding of strings as ones and zeros (bits).

A	0	C	1010	E	1100	G	1110
B	100	D	1011	F	1101	H	1111



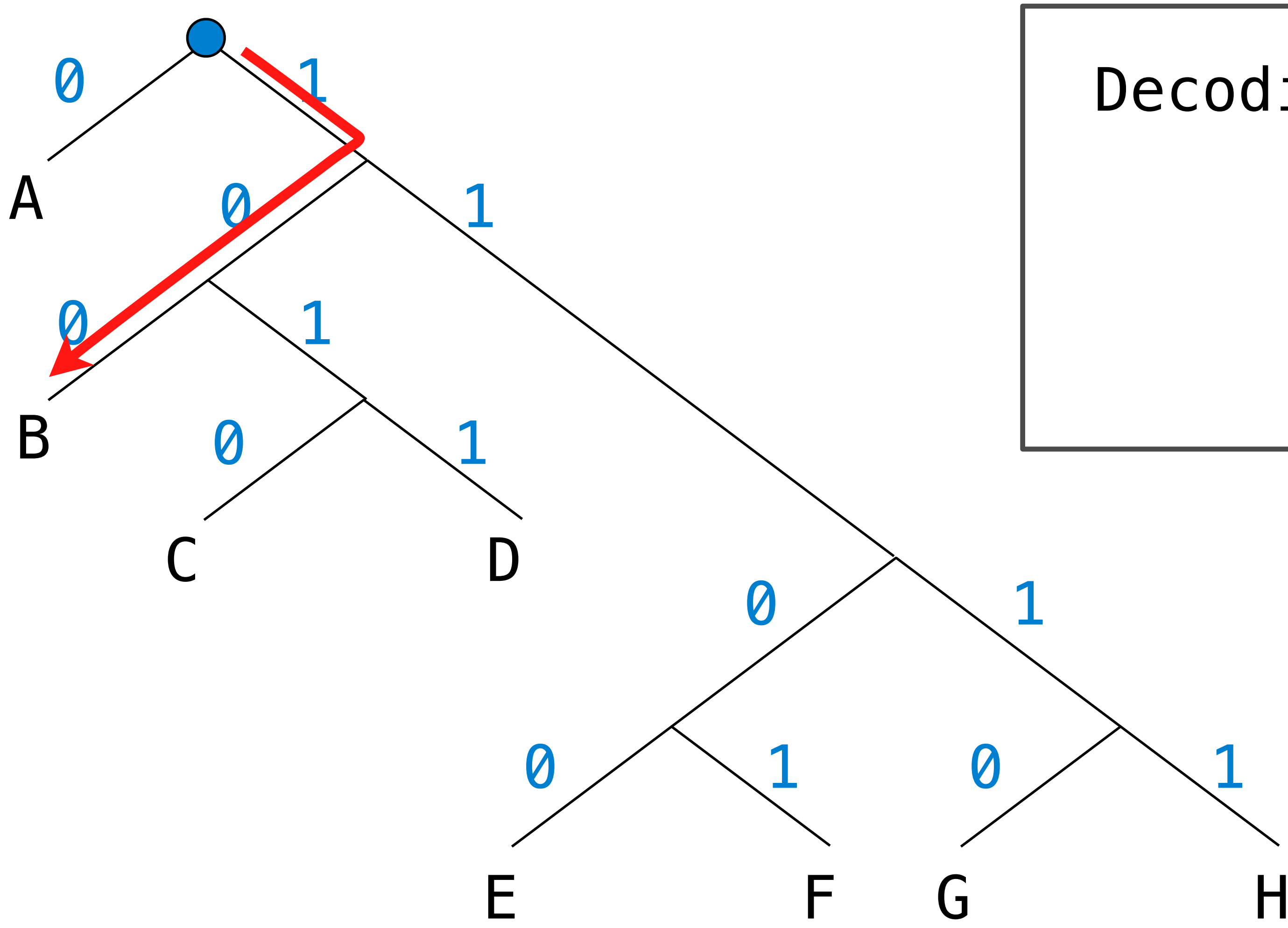
Decoding a sequence of bits:

1 0 0 0 1 0 1 0

Homework: Huffman Encoding Trees

Efficient encoding of strings as ones and zeros (bits).

A 0	C 1010	E 1100	G 1110
B 100	D 1011	F 1101	H 1111



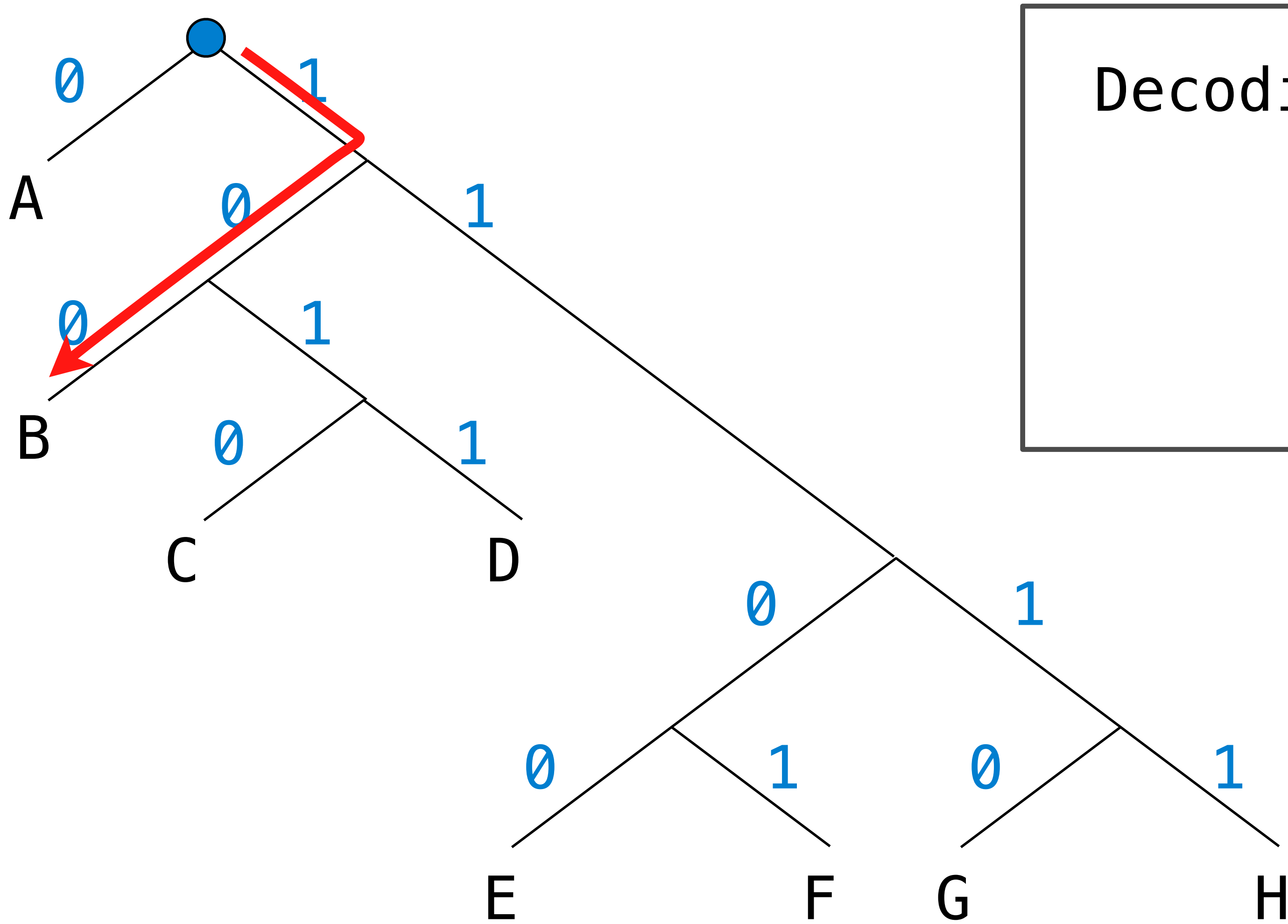
Decoding a sequence of bits:

1 0 0 0 1 0 1 0

Homework: Huffman Encoding Trees

Efficient encoding of strings as ones and zeros (bits).

A 0	C 1010	E 1100	G 1110
B 100	D 1011	F 1101	H 1111



Decoding a sequence of bits:

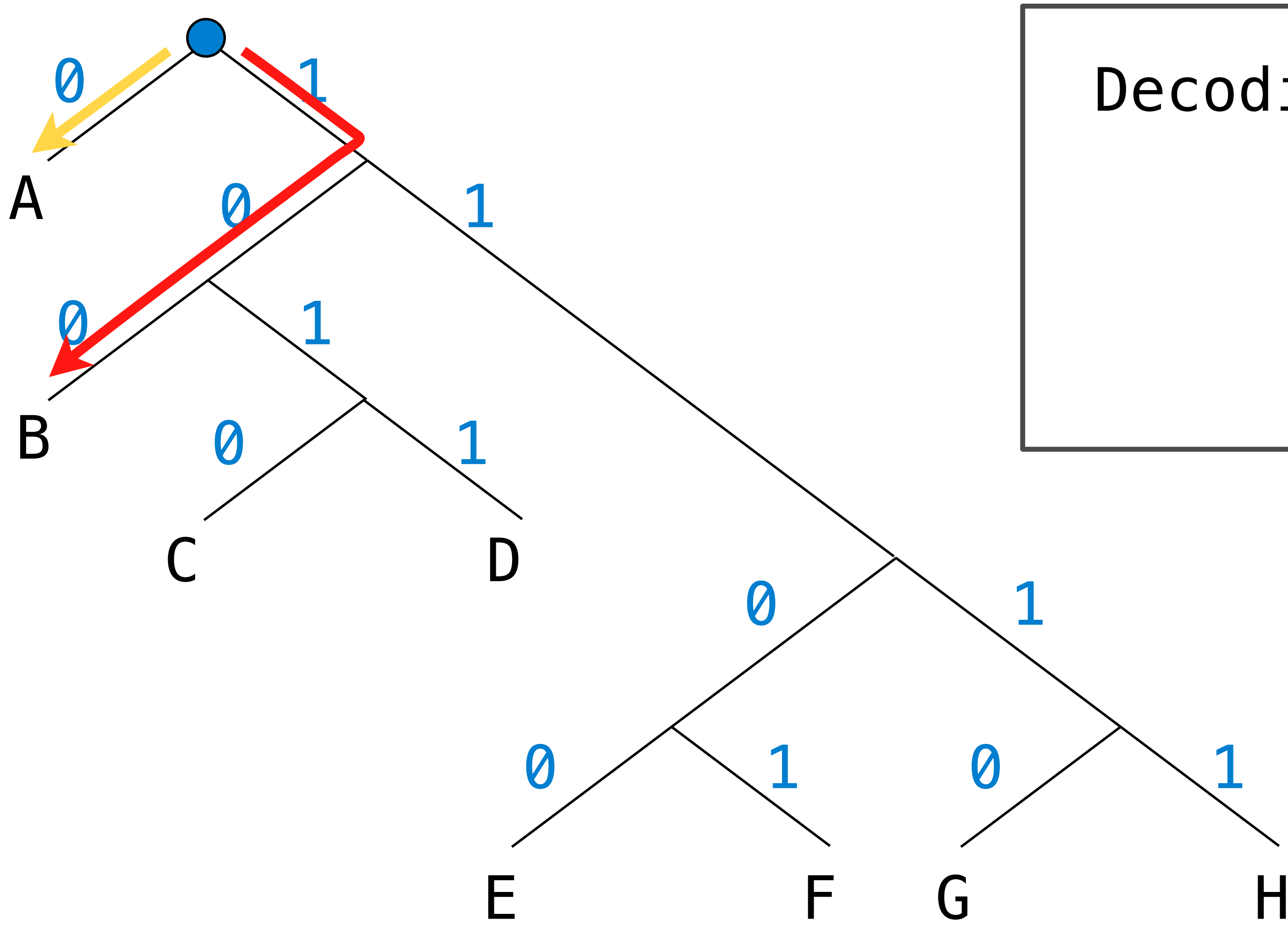
1 0 0 0 1 0 1 0

B

Homework: Huffman Encoding Trees

Efficient encoding of strings as ones and zeros (bits).

A 0	C 1010	E 1100	G 1110
B 100	D 1011	F 1101	H 1111



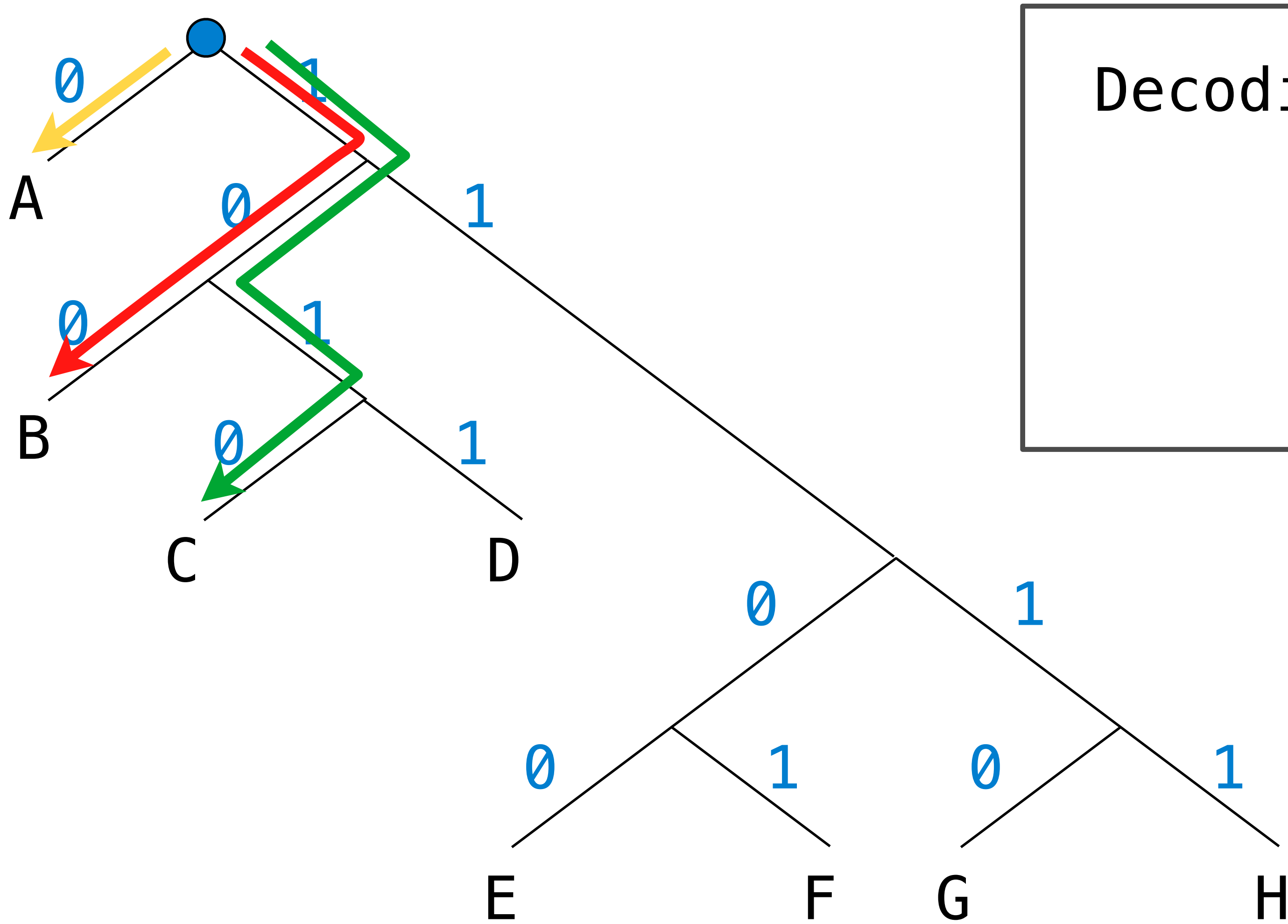
Decoding a sequence of bits:

$\underbrace{1\ 0\ 0}_{B}\ \underbrace{0}_{A}\ 1\ 0\ 1\ 0$

Homework: Huffman Encoding Trees

Efficient encoding of strings as ones and zeros (bits).

A 0	C 1010	E 1100	G 1110
B 100	D 1011	F 1101	H 1111



Decoding a sequence of bits:

1 0 0 0 1 0 1 0

B A C

Logo Refresher

Logo Refresher

Data types: Words and sentences (immutable sequences)

Logo Refresher

Data types: Words and sentences (immutable sequences)

Syntactic forms: Call expressions, literals, and to-statements

Logo Refresher

Data types: Words and sentences (immutable sequences)

Syntactic forms: Call expressions, literals, and to-statements

```
? print sum 10 difference 7 3
14
```

Logo Refresher

Data types: Words and sentences (immutable sequences)

Syntactic forms: Call expressions, literals, and to-statements

```
? print sum 10 difference 7 3
14
```

Logo Refresher

Data types: Words and sentences (immutable sequences)

Syntactic forms: Call expressions, literals, and to-statements

```
? print sum 10 difference 7 3
14
```

Logo Refresher

Data types: Words and sentences (immutable sequences)

Syntactic forms: Call expressions, literals, and to-statements

```
? print sum 10 difference 7 3
14
```

Logo Refresher

Data types: Words and sentences (immutable sequences)

Syntactic forms: Call expressions, literals, and to-statements

```
? print sum 10 difference 7 3
14

? run [print sum 1 2]
3
```

Logo Refresher

Data types: Words and sentences (immutable sequences)

Syntactic forms: Call expressions, literals, and to-statements

```
? print sum 10 difference 7 3
14

? run [print sum 1 2]
3

? to double :x
> output sum :x :x
> end
```

Logo Refresher

Data types: Words and sentences (immutable sequences)

Syntactic forms: Call expressions, literals, and to-statements

```
? print sum 10 difference 7 3
14

? run [print sum 1 2]
3

? to double :x
> output sum :x :x
> end

? print double 4
8
```

Logo Interpreter Architecture

Logo Interpreter Architecture

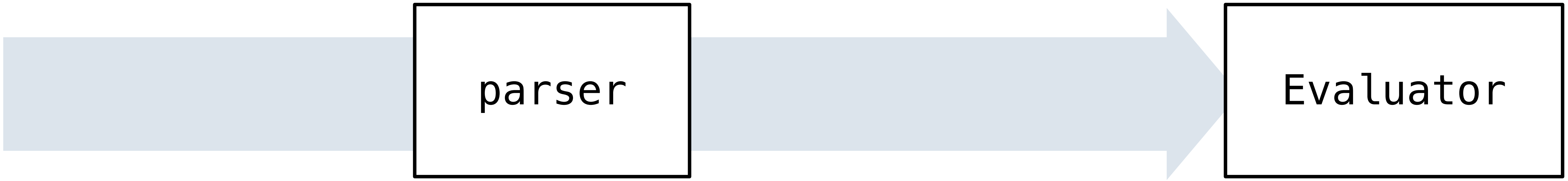
`parser`

Logo Interpreter Architecture

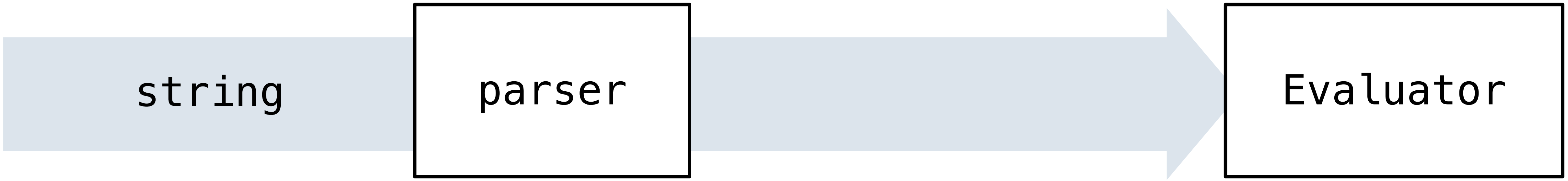
parser

Evaluator

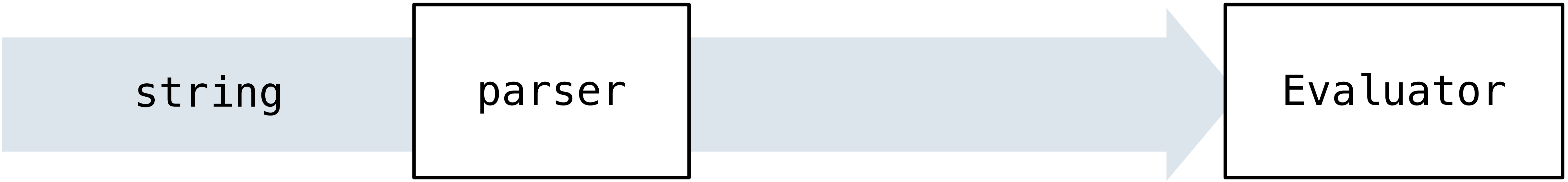
Logo Interpreter Architecture



Logo Interpreter Architecture

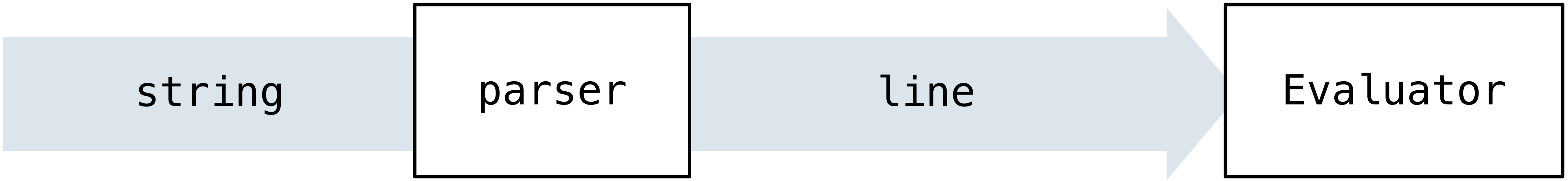


Logo Interpreter Architecture



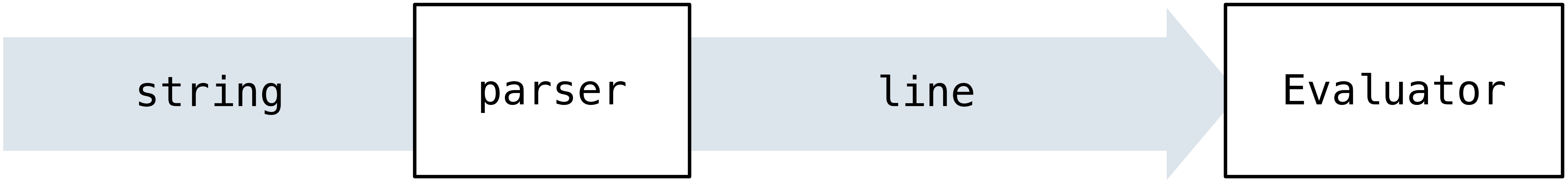
`'run [print sum 1 2]'`

Logo Interpreter Architecture




```
'run [print sum 1 2]'
```

Logo Interpreter Architecture



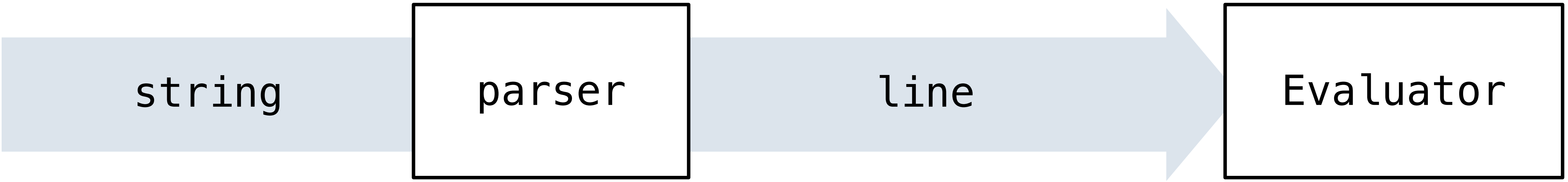

```
'run [print sum 1 2]'      ['run', ['print', 'sum', '1', '2']]
```

Logo Interpreter Architecture


```
'run [print sum 1 2]'      ['run', ['print', 'sum', '1', '2']]
```

Logo words are represented as Python strings

Logo Interpreter Architecture



`'run [print sum 1 2]'` `['run', ['print', 'sum', '1', '2']]`

Logo words are represented as Python strings

Logo sentences are represented as Python lists

Logo Interpreter Architecture

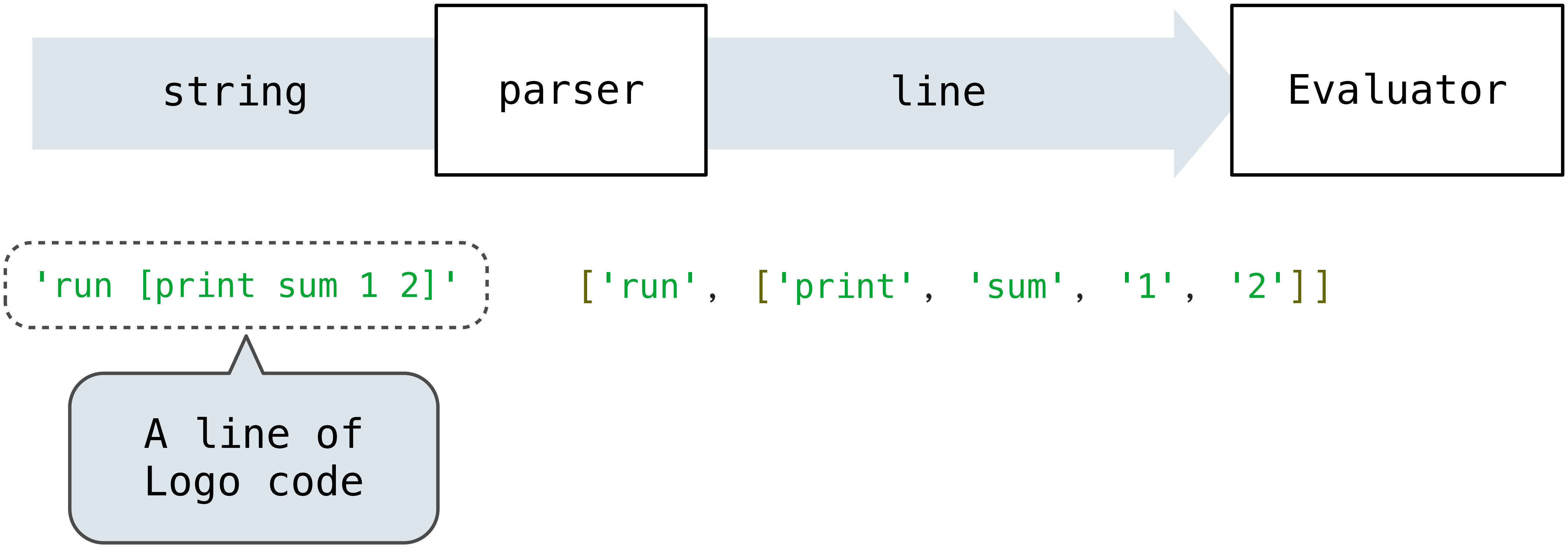

```
'run [print sum 1 2]'      ['run', ['print', 'sum', '1', '2']]
```

Logo words are represented as Python strings

Logo sentences are represented as Python lists

The Parser creates nested sentences, but **does not** build full expression trees for nested call expressions

Logo Interpreter Architecture

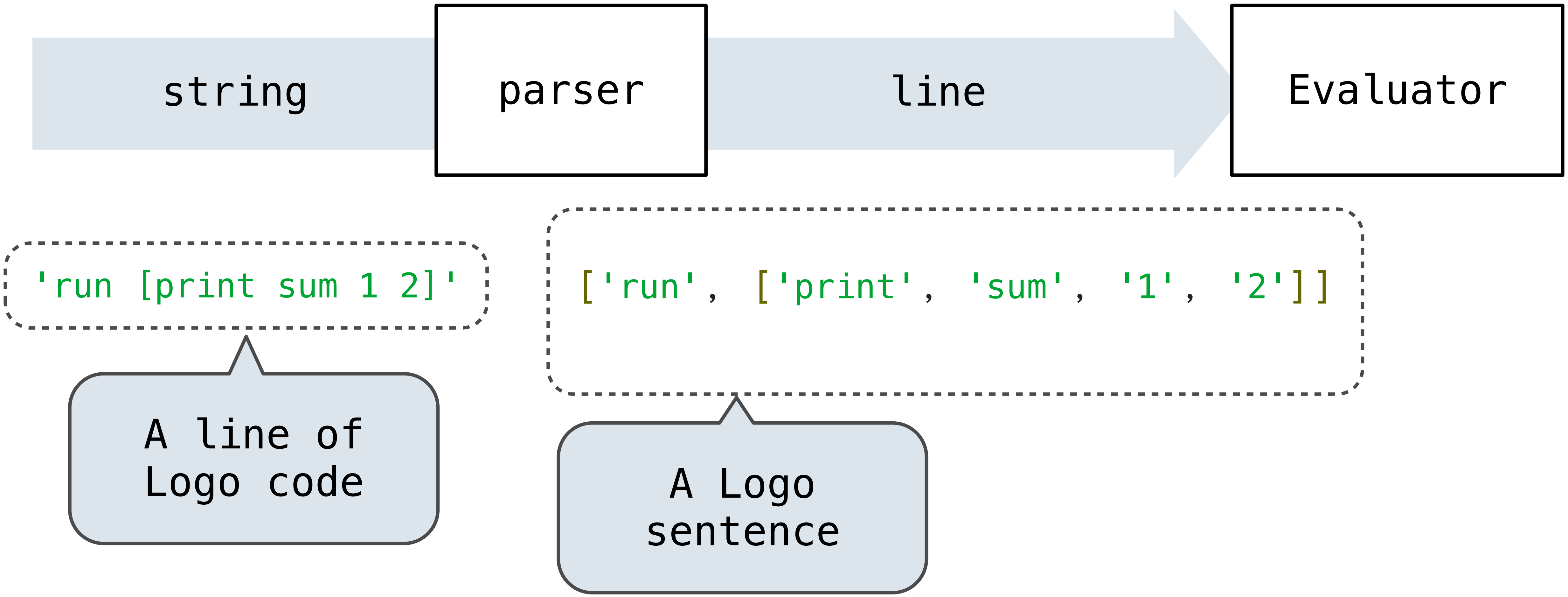


Logo words are represented as Python strings

Logo sentences are represented as Python lists

The Parser creates nested sentences, but **does not** build full expression trees for nested call expressions

Logo Interpreter Architecture

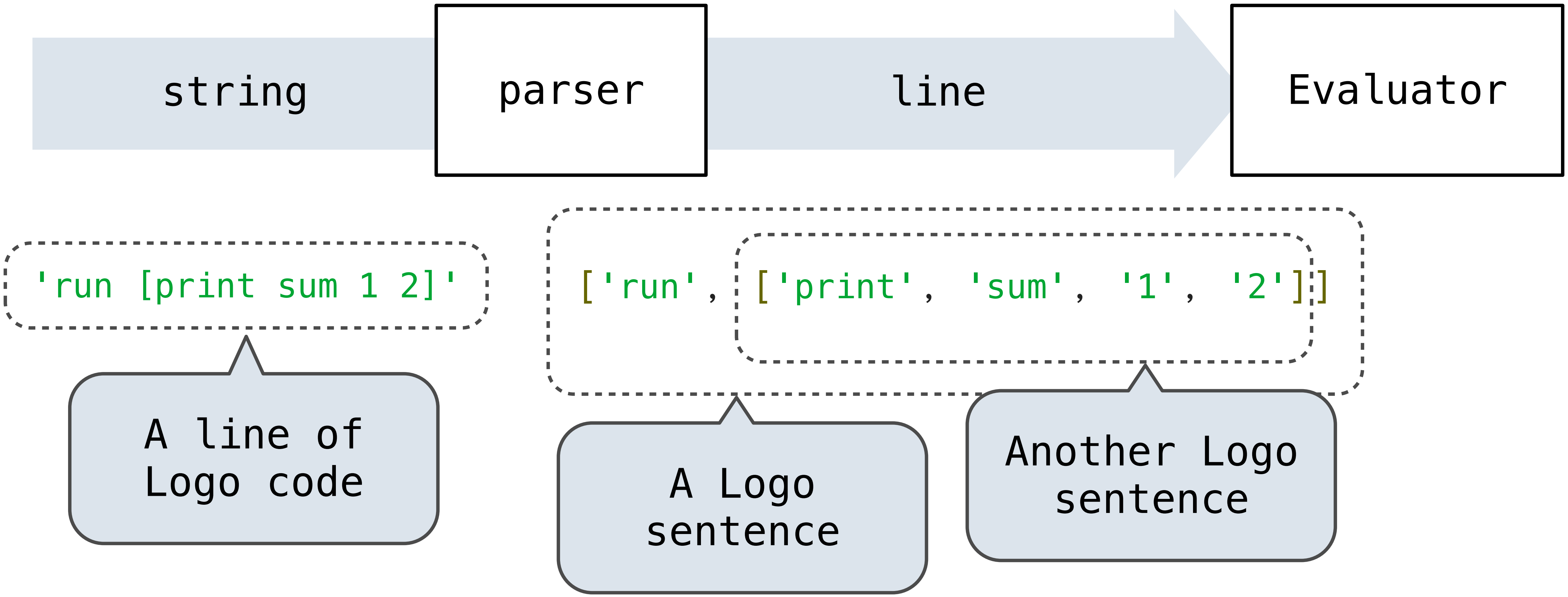


Logo words are represented as Python strings

Logo sentences are represented as Python lists

The Parser creates nested sentences, but **does not** build full expression trees for nested call expressions

Logo Interpreter Architecture

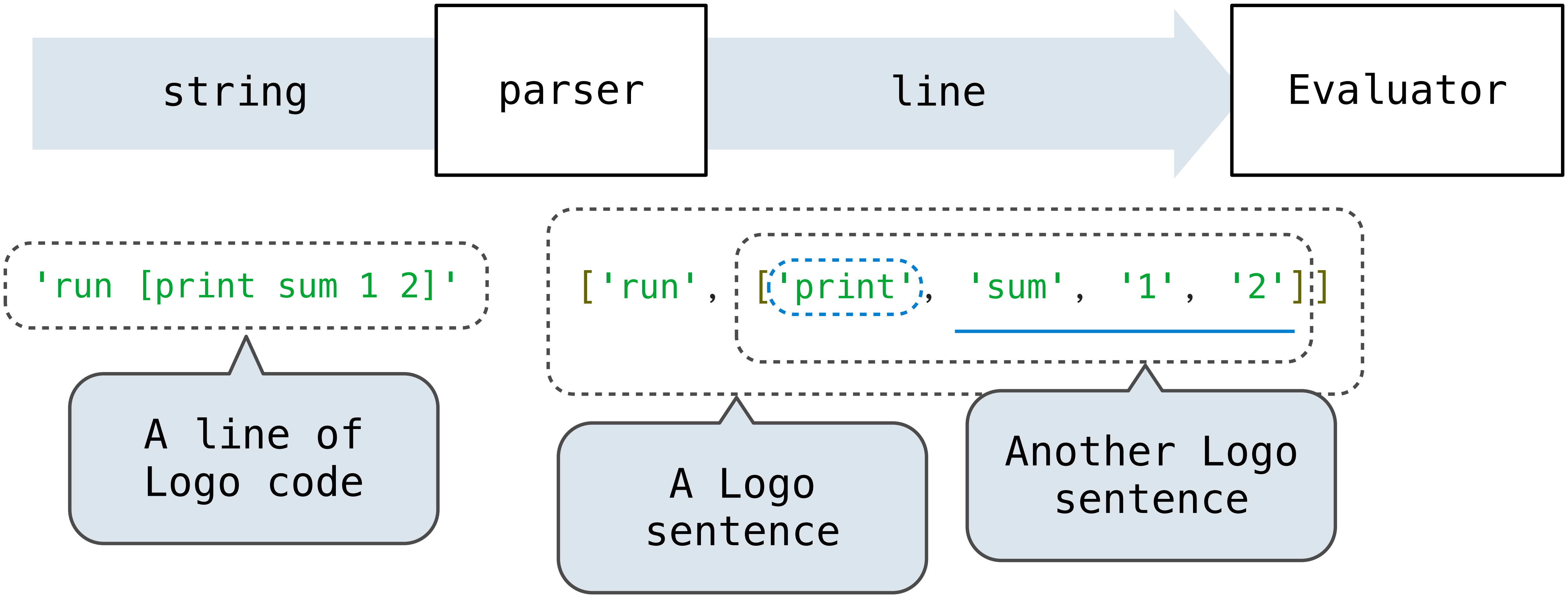


Logo words are represented as Python strings

Logo sentences are represented as Python lists

The Parser creates nested sentences, but **does not** build full expression trees for nested call expressions

Logo Interpreter Architecture

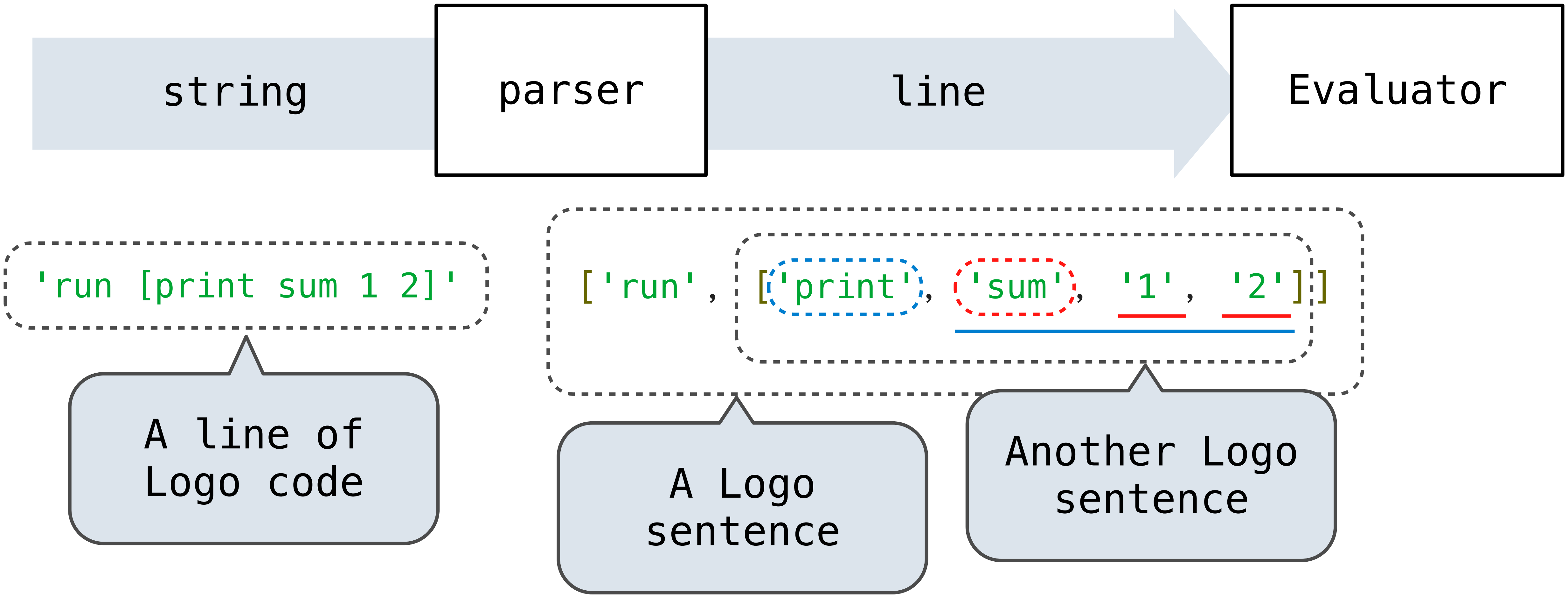


Logo words are represented as Python strings

Logo sentences are represented as Python lists

The Parser creates nested sentences, but **does not** build full expression trees for nested call expressions

Logo Interpreter Architecture



Logo words are represented as Python strings

Logo sentences are represented as Python lists

The Parser creates nested sentences, but **does not** build full expression trees for nested call expressions

Tracking Positions in Lines

Tracking Positions in Lines

```
A line is used up as it is evaluated
```

Tracking Positions in Lines

A line is used up as it is evaluated

A Buffer instance tracks how much of a line has been used up.

Tracking Positions in Lines

A line is used up as it is evaluated

A Buffer instance tracks how much of a line has been used up.

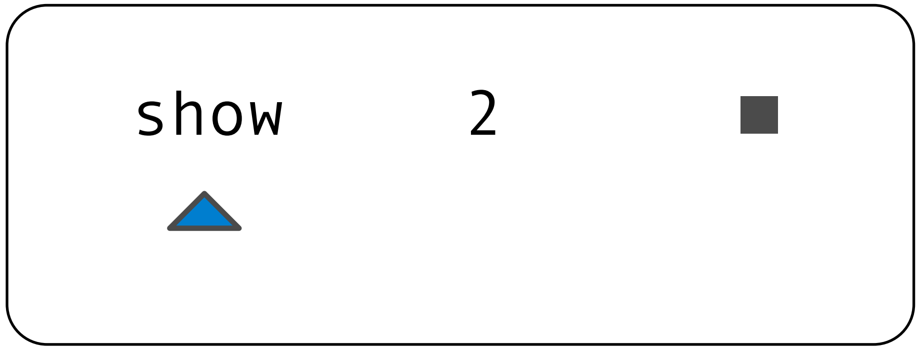
```
>>> buf = Buffer(['show', '2'])
```

Tracking Positions in Lines

A line is used up as it is evaluated

A Buffer instance tracks how much of a line has been used up.

```
>>> buf = Buffer(['show', '2'])
```

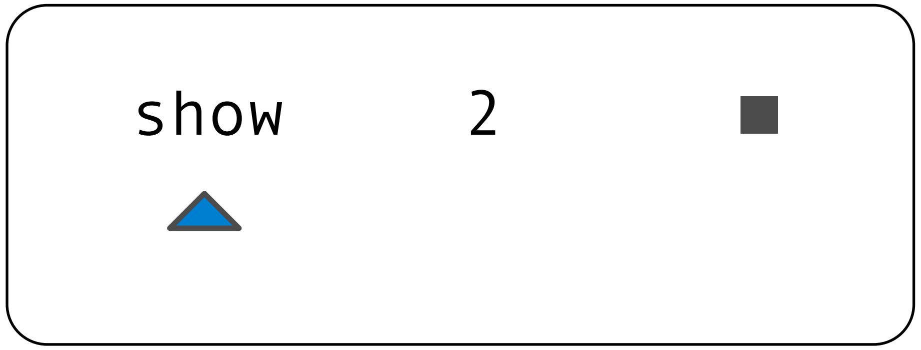


Tracking Positions in Lines

A line is used up as it is evaluated

A Buffer instance tracks how much of a line has been used up.

```
>>> buf = Buffer(['show', '2'])
>>> buf.current
'show'
```

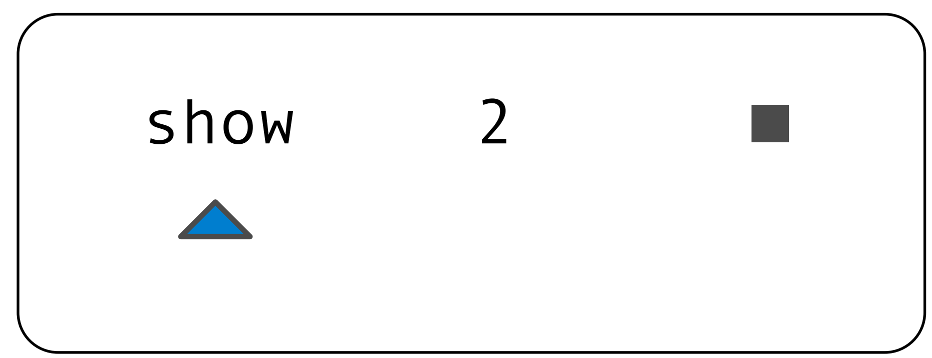


Tracking Positions in Lines

A line is used up as it is evaluated

A Buffer instance tracks how much of a line has been used up.

```
>>> buf = Buffer(['show', '2'])
>>> buf.current
'show'
>>> print(buf)
[  >> show, 2 ]
```

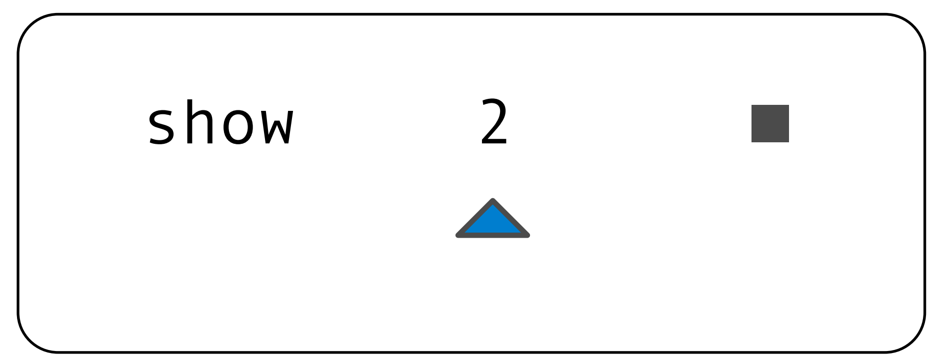


Tracking Positions in Lines

A line is used up as it is evaluated

A Buffer instance tracks how much of a line has been used up.

```
>>> buf = Buffer(['show', '2'])
>>> buf.current
'show'
>>> print(buf)
[  >> show, 2 ]

>>> buf.pop()
'show'
```

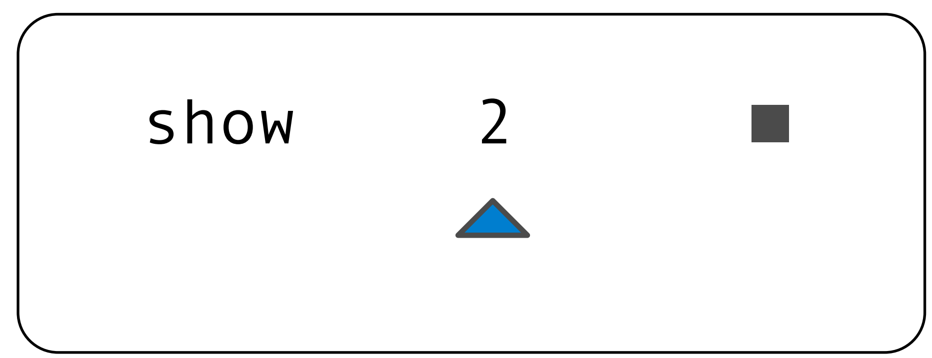


Tracking Positions in Lines

A line is used up as it is evaluated

A Buffer instance tracks how much of a line has been used up.

```
>>> buf = Buffer(['show', '2'])
>>> buf.current
'show'
>>> print(buf)
[  >> show, 2 ]

>>> buf.pop()
'show'
>>> print(buf)
[ show >> 2 ]
```

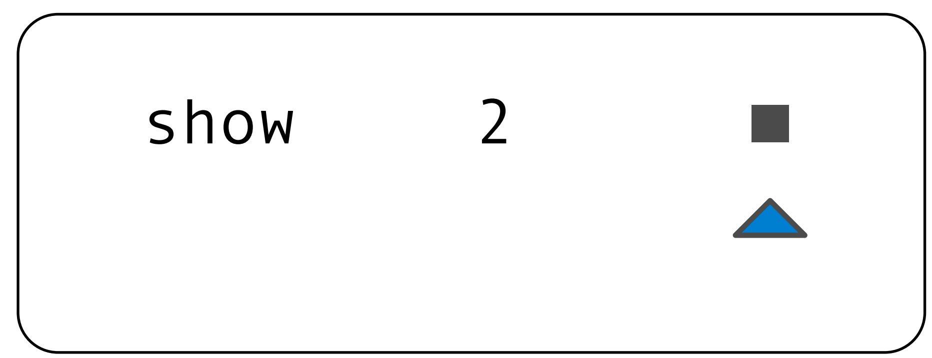


Tracking Positions in Lines

A line is used up as it is evaluated

A Buffer instance tracks how much of a line has been used up.

```
>>> buf = Buffer(['show', '2'])
>>> buf.current
'show'
>>> print(buf)
[  >> show, 2 ]

>>> buf.pop()
'show'
>>> print(buf)
[ show >> 2 ]

>>> buf.pop()
'2'
```

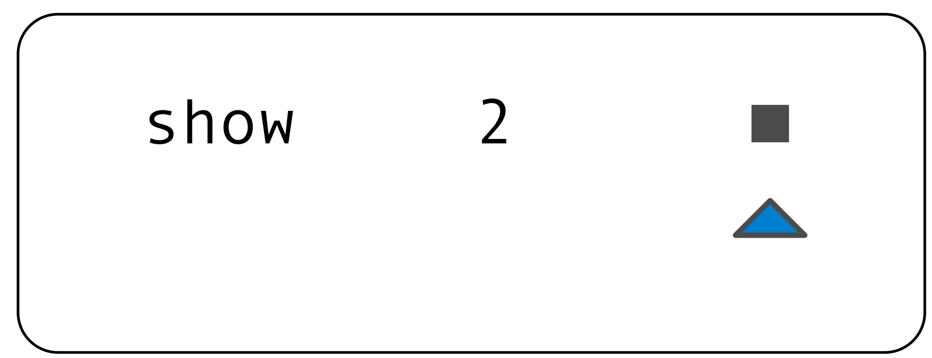


Tracking Positions in Lines

A line is used up as it is evaluated

A Buffer instance tracks how much of a line has been used up.

```
>>> buf = Buffer(['show', '2'])
>>> buf.current
'show'
>>> print(buf)
[  >> show, 2 ]

>>> buf.pop()
'show'
>>> print(buf)
[ show >> 2 ]

>>> buf.pop()
'2'
```



Demo

Evaluating Lines

Evaluating Lines

`Evaluating a line of Logo involves evaluating each expression`

Evaluating Lines

Evaluating a line of Logo involves evaluating each expression

Evaluate a line

`eval_line`

Evaluating Lines

Evaluating a line of Logo involves evaluating each expression

Evaluate a line — `eval_line`

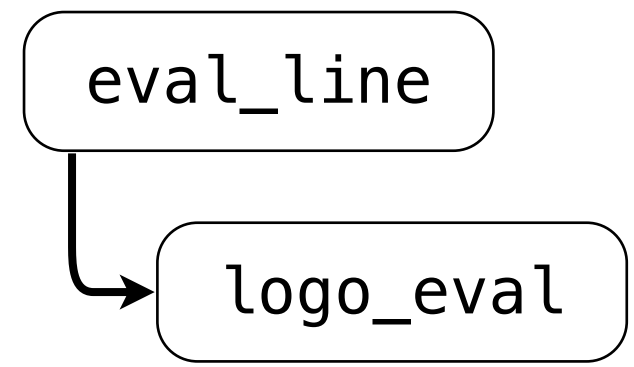
Evaluate the next expression — `logo_eval`

Evaluating Lines

Evaluating a line of Logo involves evaluating each expression

Evaluate a line

Evaluate the
next expression

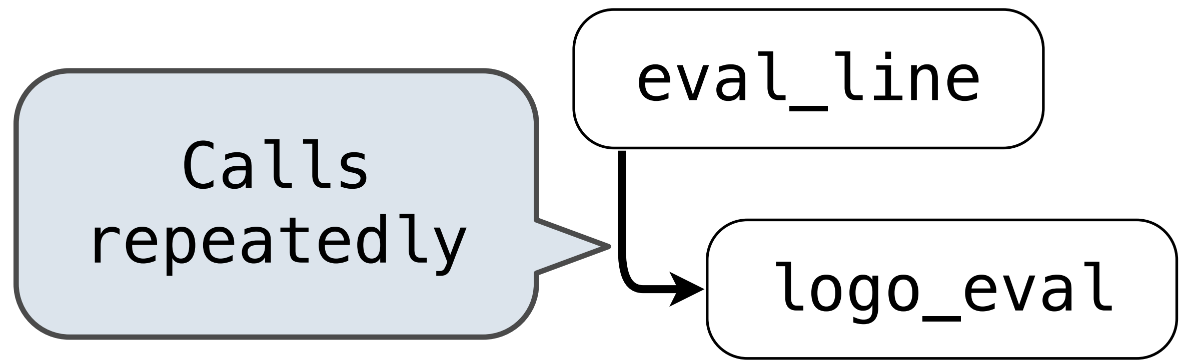


Evaluating Lines

Evaluating a line of Logo involves evaluating each expression

Evaluate a line

Evaluate the
next expression

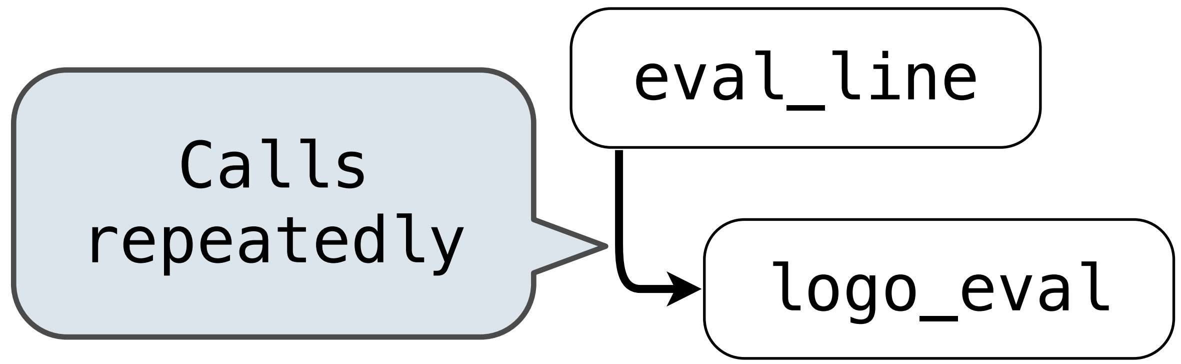


Evaluating Lines

Evaluating a line of Logo involves evaluating each expression

Evaluate a line

Evaluate the next expression



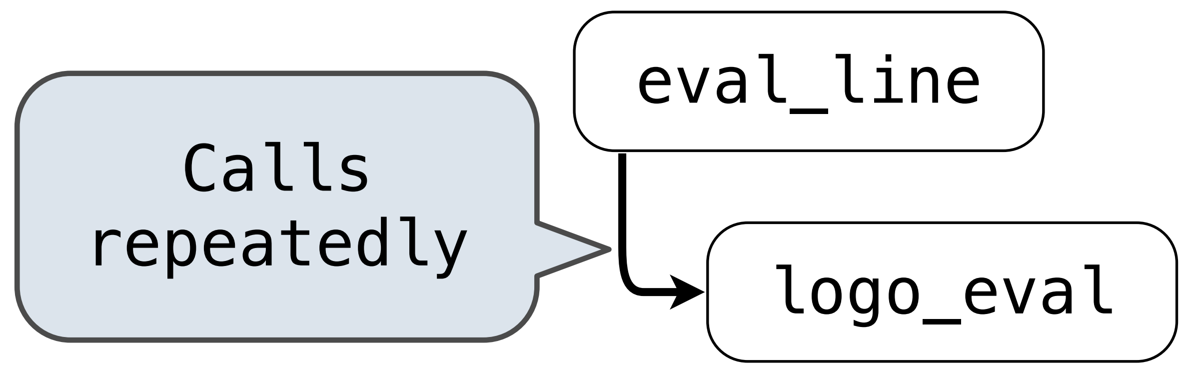

```
? print 1 print 2
1
2
```

Evaluating Lines

Evaluating a line of Logo involves evaluating each expression

Evaluate a line

Evaluate the
next expression




```
? print 1 print 2
1
2
```

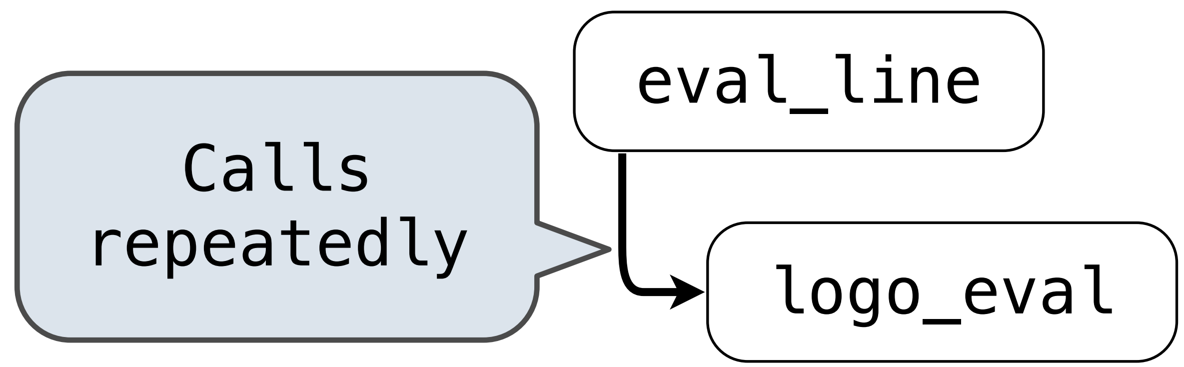
logo_eval	Argument	Effect
first call		
second call		

Evaluating Lines

Evaluating a line of Logo involves evaluating each expression

Evaluate a line

Evaluate the next expression




```
? print 1 print 2
1
2
```

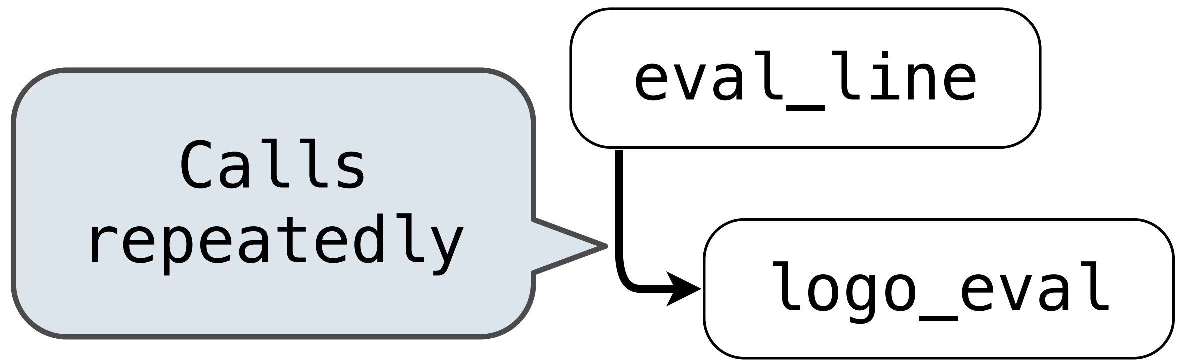
logo_eval	**Argument**	**Effect**
first call	[>> print, 1, print, 2]	
second call		

Evaluating Lines

Evaluating a line of Logo involves evaluating each expression

Evaluate a line

Evaluate the
next expression




```
? print 1 print 2
1
2
```

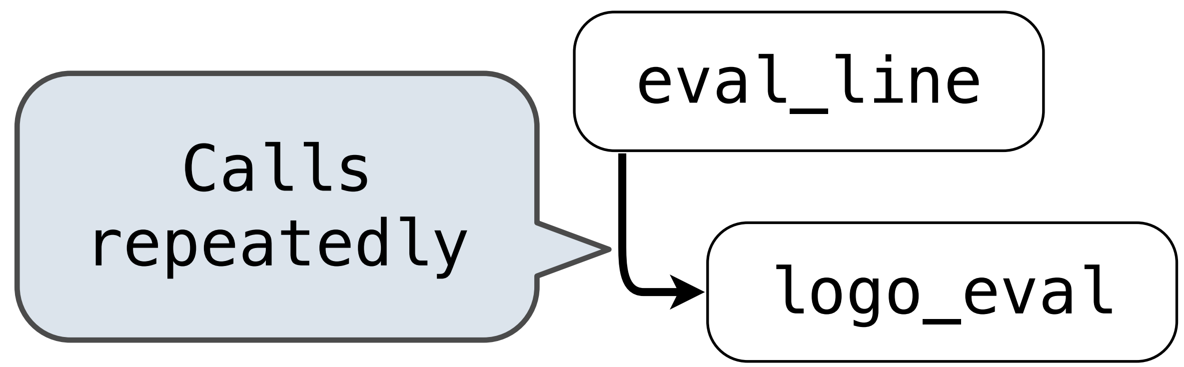
logo_eval	**Argument**	**Effect**
first call	[>> print, 1, print, 2]	prints 1, returns None
second call		

Evaluating Lines

Evaluating a line of Logo involves evaluating each expression

Evaluate a line

Evaluate the
next expression




```
? print 1 print 2
1
2
```

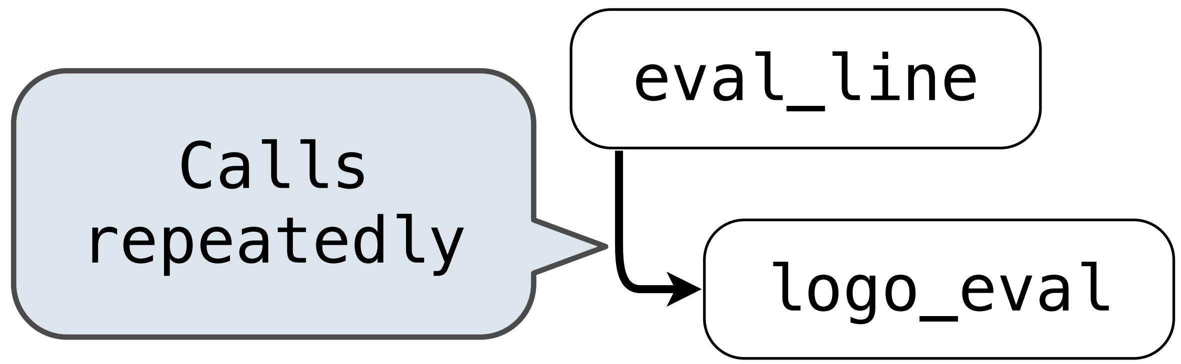
logo_eval	**Argument**	**Effect**
first call	[>> print, 1, print, 2]	prints 1, returns None
second call	[print, 1 >> print, 2]	

Evaluating Lines

Evaluating a line of Logo involves evaluating each expression

Evaluate a line

Evaluate the next expression




```
? print 1 print 2
1
2
```

logo_eval	Argument	Effect
first call	[>> print, 1, print, 2]	prints 1, returns None
second call	[print, 1 >> print, 2]	prints 2, returns None

Logo Evaluation

Logo Evaluation

The logo_eval function dispatches on expression form:

Logo Evaluation

The logo_eval function dispatches on expression form:

- A **primitive expression** is a word that can be interpreted as a number, True, or False. Primitives are self evaluating.

Logo Evaluation

The logo_eval function dispatches on expression form:

- A **primitive expression** is a word that can be interpreted as a number, True, or False. Primitives are self evaluating.
- A **variable** is looked up in the current environment.

Logo Evaluation

The logo_eval function dispatches on expression form:

- A **primitive expression** is a word that can be interpreted as a number, True, or False. Primitives are self evaluating.
- A **variable** is looked up in the current environment.
- A **procedure definition** creates a new user-defined procedure.

Logo Evaluation

The logo_eval function dispatches on expression form:

- A **primitive expression** is a word that can be interpreted as a number, True, or False. Primitives are self evaluating.
- A **variable** is looked up in the current environment.
- A **procedure definition** creates a new user-defined procedure.
- A **quoted expression** evaluates to the text of the quotation, which is a string without the preceding quote. Sentences are quoted and evaluate to themselves.

Logo Evaluation

The logo_eval function dispatches on expression form:

- A **primitive expression** is a word that can be interpreted as a number, True, or False. Primitives are self evaluating.
- A **variable** is looked up in the current environment.
- A **procedure definition** creates a new user-defined procedure.
- A **quoted expression** evaluates to the text of the quotation, which is a string without the preceding quote. Sentences are quoted and evaluate to themselves.
- A **call expression** is evaluated with apply_procedure.

Logo Evaluation

The logo_eval function dispatches on expression form:

- A **primitive expression** is a word that can be interpreted as a number, True, or False. Primitives are self evaluating.
- A **variable** is looked up in the current environment.
- A **procedure definition** creates a new user-defined procedure.
- A **quoted expression** evaluates to the text of the quotation, which is a string without the preceding quote. Sentences are quoted and evaluate to themselves.
- A **call expression** is evaluated with apply_procedure.

```
def logo_eval(line, env):
```

Logo Evaluation

The logo_eval function dispatches on expression form:

- A **primitive expression** is a word that can be interpreted as a number, True, or False. Primitives are self evaluating.
- A **variable** is looked up in the current environment.
- A **procedure definition** creates a new user-defined procedure.
- A **quoted expression** evaluates to the text of the quotation, which is a string without the preceding quote. Sentences are quoted and evaluate to themselves.
- A **call expression** is evaluated with apply_procedure.

```
def logo_eval(line, env):
    """Evaluate the first expression in a line."""
```

Logo Evaluation

The logo_eval function dispatches on expression form:

- A **primitive expression** is a word that can be interpreted as a number, True, or False. Primitives are self evaluating.
- A **variable** is looked up in the current environment.
- A **procedure definition** creates a new user-defined procedure.
- A **quoted expression** evaluates to the text of the quotation, which is a string without the preceding quote. Sentences are quoted and evaluate to themselves.
- A **call expression** is evaluated with apply_procedure.

```
def logo_eval(line, env):
    """Evaluate the first expression in a line."""
    token = line.pop()
```

Logo Evaluation

The logo_eval function dispatches on expression form:

- A **primitive expression** is a word that can be interpreted as a number, True, or False. Primitives are self evaluating.
- A **variable** is looked up in the current environment.
- A **procedure definition** creates a new user-defined procedure.
- A **quoted expression** evaluates to the text of the quotation, which is a string without the preceding quote. Sentences are quoted and evaluate to themselves.
- A **call expression** is evaluated with apply_procedure.

```
def logo_eval(line, env):
    """Evaluate the first expression in a line."""
    token = line.pop()
```

The expression form can be inferred from the first token

Logo Evaluation

The logo_eval function dispatches on expression form:

- A **primitive expression** is a word that can be interpreted as a number, True, or False. Primitives are self evaluating.
- A **variable** is looked up in the current environment.
- A **procedure definition** creates a new user-defined procedure.
- A **quoted expression** evaluates to the text of the quotation, which is a string without the preceding quote. Sentences are quoted and evaluate to themselves.
- A **call expression** is evaluated with apply_procedure.

```
def logo_eval(line, env):
    """Evaluate the first expression in a line."""
    token = line.pop()
    if isprimitive(token):
```

The expression form can be inferred from the first token

Logo Evaluation

The logo_eval function dispatches on expression form:

- A **primitive expression** is a word that can be interpreted as a number, True, or False. Primitives are self evaluating.
- A **variable** is looked up in the current environment.
- A **procedure definition** creates a new user-defined procedure.
- A **quoted expression** evaluates to the text of the quotation, which is a string without the preceding quote. Sentences are quoted and evaluate to themselves.
- A **call expression** is evaluated with apply_procedure.

```
def logo_eval(line, env):
    """Evaluate the first expression in a line."""
    token = line.pop()
    if isprimitive(token):
        return token
```

The expression form can be inferred from the first token

Logo Evaluation

The logo_eval function dispatches on expression form:

- A **primitive expression** is a word that can be interpreted as a number, True, or False. Primitives are self evaluating.
- A **variable** is looked up in the current environment.
- A **procedure definition** creates a new user-defined procedure.
- A **quoted expression** evaluates to the text of the quotation, which is a string without the preceding quote. Sentences are quoted and evaluate to themselves.
- A **call expression** is evaluated with apply_procedure.

The expression form can be inferred from the first token

```
def logo_eval(line, env):
    """Evaluate the first expression in a line."""
    token = line.pop()
    if isprimitive(token):
        return token
    elif isvariable(token):
```

Logo Evaluation

The logo_eval function dispatches on expression form:

- A **primitive expression** is a word that can be interpreted as a number, True, or False. Primitives are self evaluating.
- A **variable** is looked up in the current environment.
- A **procedure definition** creates a new user-defined procedure.
- A **quoted expression** evaluates to the text of the quotation, which is a string without the preceding quote. Sentences are quoted and evaluate to themselves.
- A **call expression** is evaluated with apply_procedure.

The expression form can be inferred from the first token

```
def logo_eval(line, env):
    """Evaluate the first expression in a line."""
    token = line.pop()
    if isprimitive(token):
        return token
    elif isvariable(token):
        ...
```

Evaluating Call Expressions

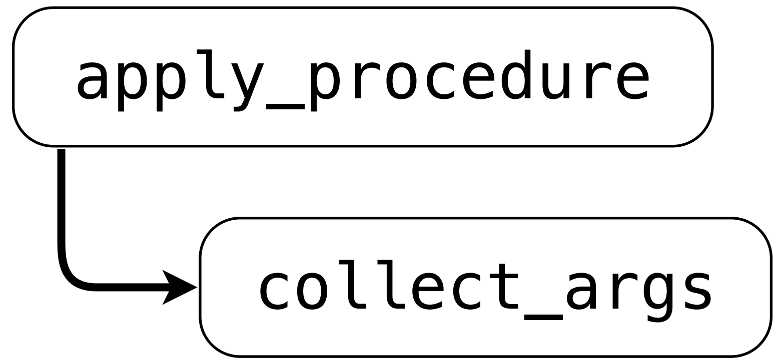
Evaluating Call Expressions

Apply a named procedure `apply_procedure`

Evaluating Call Expressions

Apply a named procedure

Evaluate *n* operands

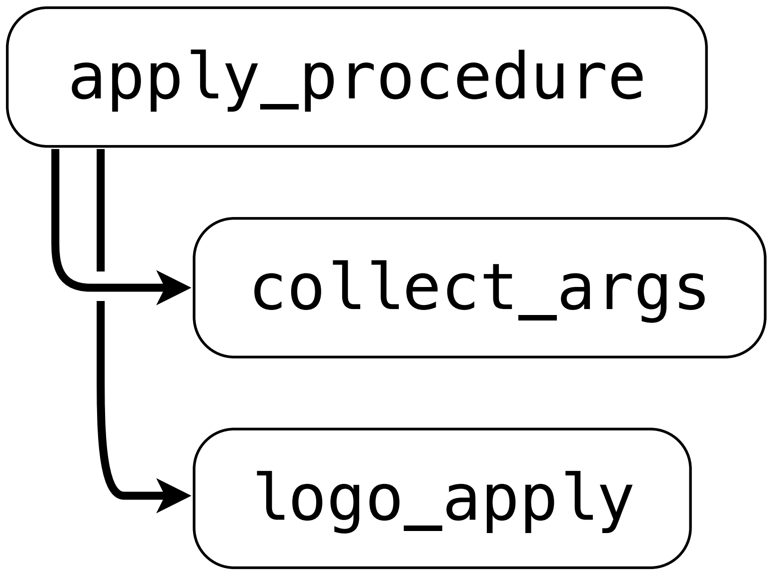


Evaluating Call Expressions

Apply a named procedure

Evaluate *n* operands

Apply a procedure to
a sequence of arguments

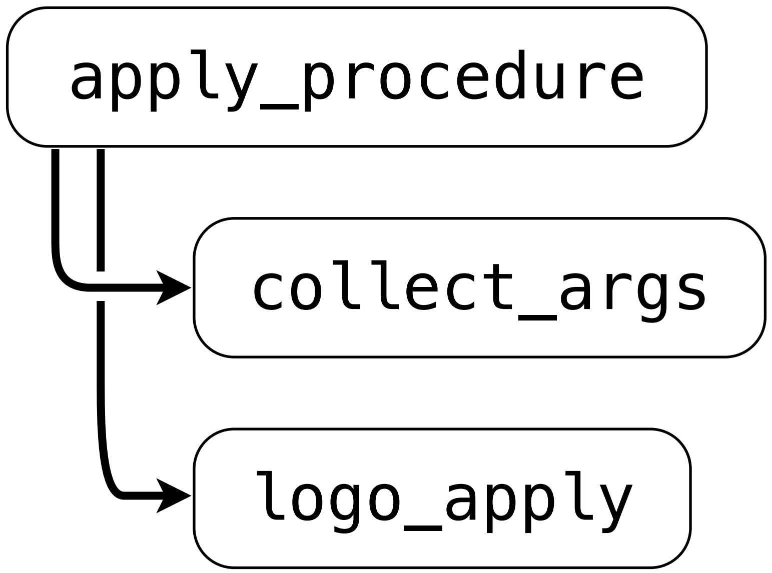


Evaluating Call Expressions

Apply a named procedure

Evaluate *n* operands

Apply a procedure to
a sequence of arguments



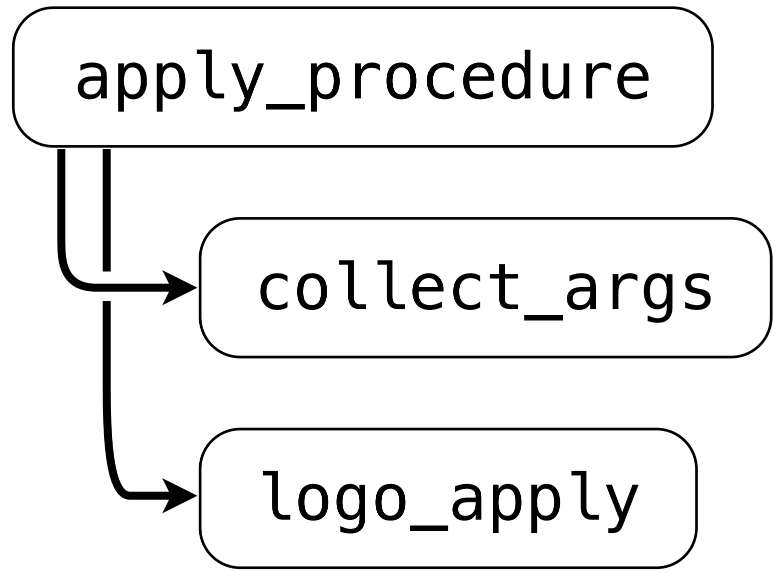
Return the output value

Evaluating Call Expressions

Apply a named procedure

Evaluate *n* operands

Apply a procedure to
a sequence of arguments



Return the output value

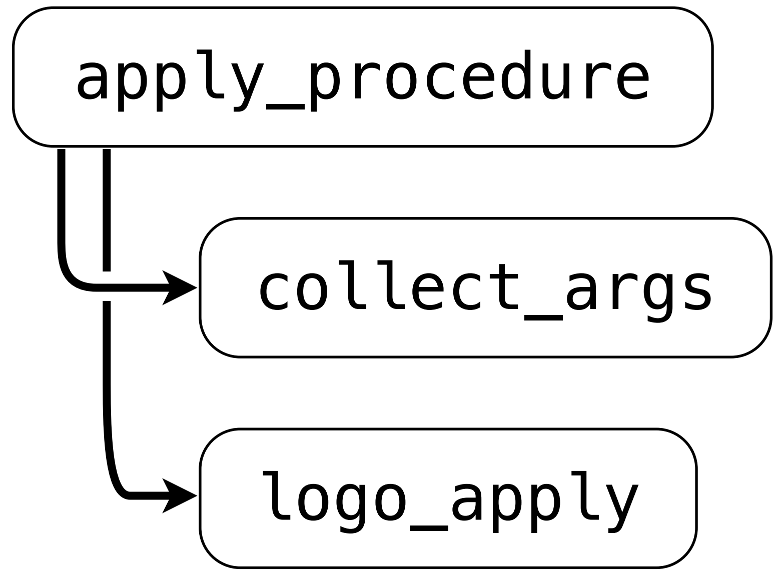
Return *n* arguments

Evaluating Call Expressions

Apply a named procedure

Evaluate *n* operands

Apply a procedure to
a sequence of arguments



Return the output value

Return *n* arguments

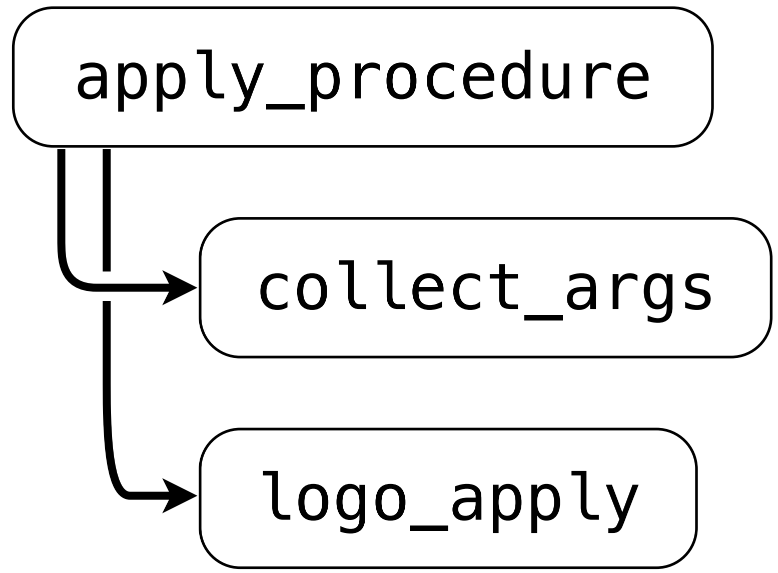
Return the output value

Evaluating Call Expressions

Apply a named procedure

Evaluate n operands

Apply a procedure to
a sequence of arguments



Return the output value

Return n arguments

Return the output value

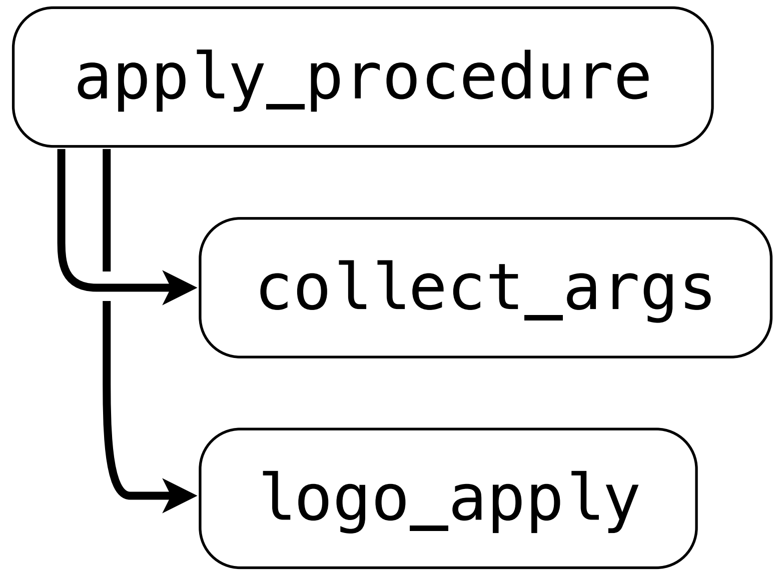
```
[ print >> 2 ]
```

Evaluating Call Expressions

Apply a named procedure

Evaluate *n* operands

Apply a procedure to
a sequence of arguments



Return the output value

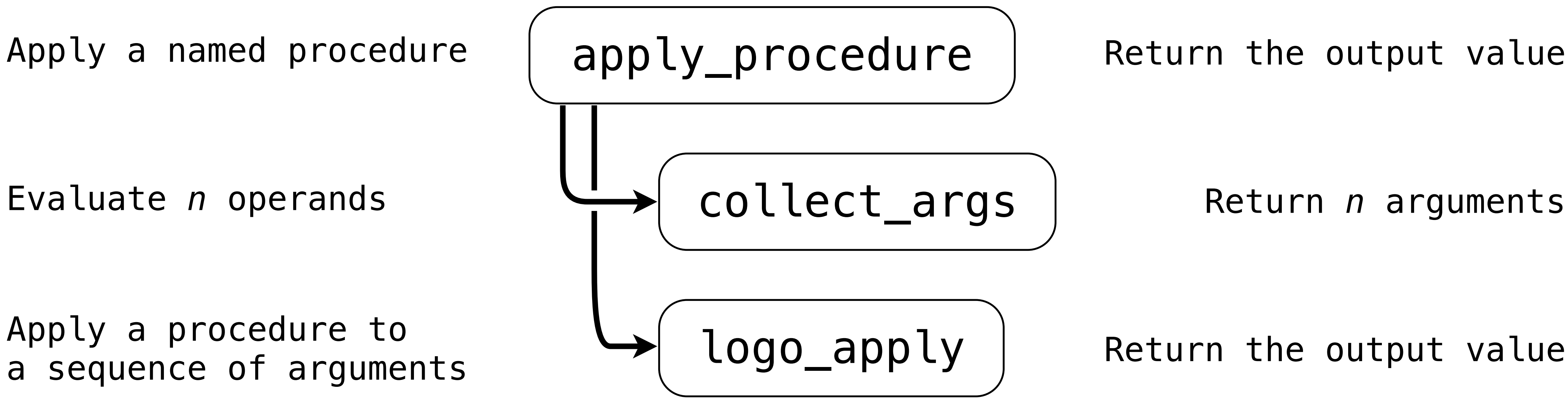
Return *n* arguments

Return the output value

```
[ print >> 2 ]
```

Popped by logo_eval

Evaluating Call Expressions




```
[ print >> 2 ]
```

Popped by logo_eval

1. Collect 1 argument via logo_eval (collect_args)

Evaluating Call Expressions

Apply a named procedure — **apply_procedure** — Return the output value

Evaluate *n* operands — **collect_args** — Return *n* arguments

Apply a procedure to a sequence of arguments — **logo_apply** — Return the output value

```
[ print >> 2 ]
```

Popped by logo_eval

```
[ print, 2 >> ]
```

1. Collect 1 argument via logo_eval (collect_args)

Evaluating Call Expressions

Apply a named procedure — **apply_procedure** — Return the output value

Evaluate *n* operands — **collect_args** — Return *n* arguments

Apply a procedure to a sequence of arguments — **logo_apply** — Return the output value

```
[ print >> 2 ]
```

Popped by logo_eval

```
[ print, 2 >> ]
```

Popped by logo_eval also (recursive call)

1. Collect 1 argument via logo_eval (collect_args)

Evaluating Call Expressions

Apply a named procedure — `apply_procedure` — Return the output value

Evaluate *n* operands — `collect_args` — Return *n* arguments

Apply a procedure to a sequence of arguments — `logo_apply` — Return the output value

```
[ print >> 2 ]
```

Popped by logo_eval

```
[ print, 2 >> ]
```

Popped by logo_eval also (recursive call)

1. Collect 1 argument via logo_eval (collect_args)

2. Apply print procedure to the argument '2' (logo_apply)

Procedures

Procedures

```
class Procedure():
```

Procedures

```
class Procedure():
    def __init__(self, name, arg_count, body, isprimitive=False,
                 needs_env=False, formal_params=None):
```

Procedures

```
class Procedure():
    def __init__(self, name, arg_count, body, isprimitive=False,
                 needs_env=False, formal_params=None):
        self.name = name
        self.arg_count = arg_count
        self.body = body
        self.isprimitive = isprimitive
        self.needs_env = needs_env
        self.formal_params = formal_params
```

Procedures

```
class Procedure():
    def __init__(self, name, arg_count, body, isprimitive=False,
                 needs_env=False, formal_params=None):
        self.name = name
        self.arg_count = arg_count
        self.body = body
        self.isprimitive = isprimitive
        self.needs_env = needs_env
        self.formal_params = formal_params

def logo_apply(proc, args):
    """Apply a Logo procedure to a list of arguments."""
```

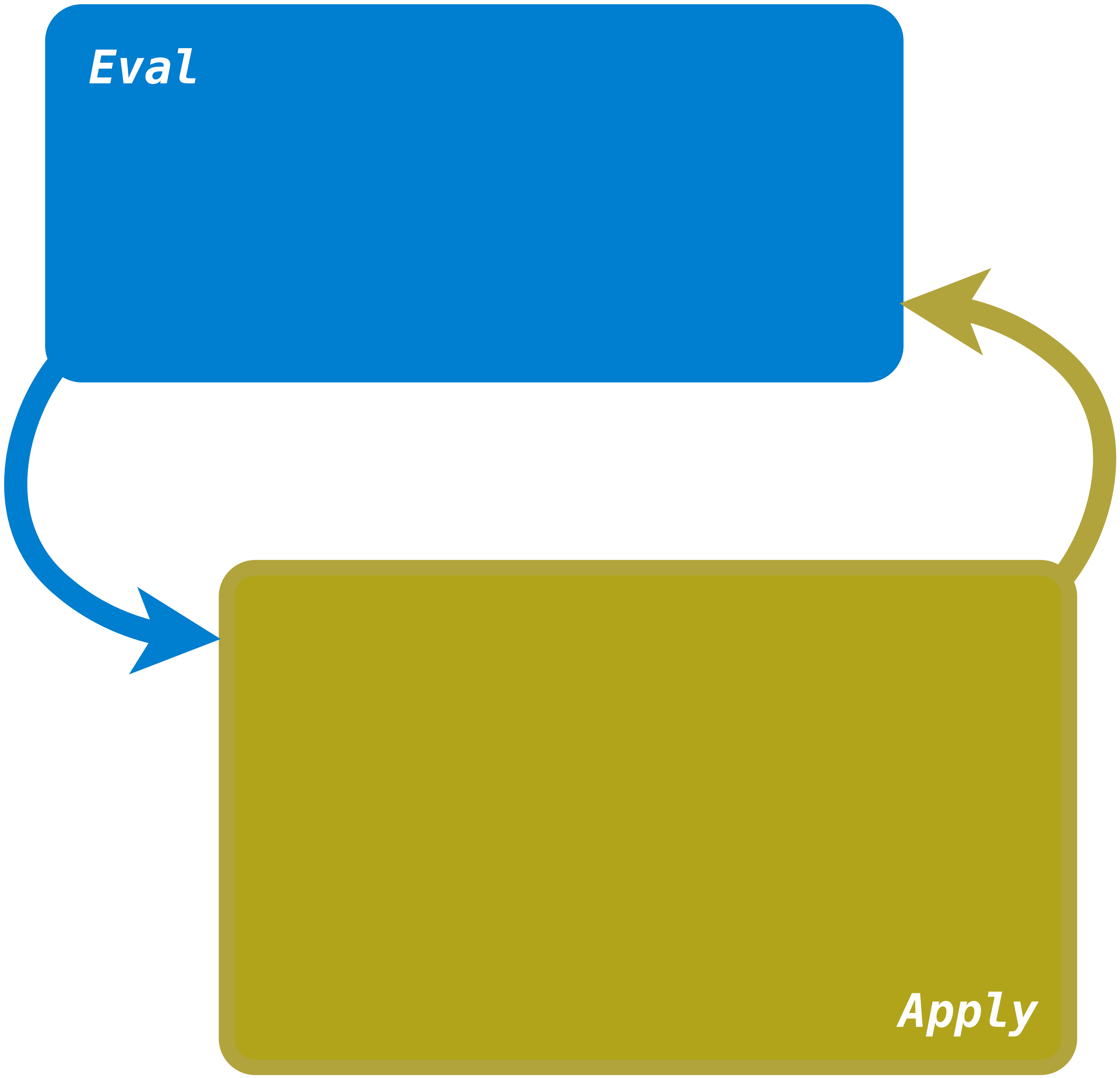
Procedures

```
class Procedure():
    def __init__(self, name, arg_count, body, isprimitive=False,
                 needs_env=False, formal_params=None):
        self.name = name
        self.arg_count = arg_count
        self.body = body
        self.isprimitive = isprimitive
        self.needs_env = needs_env
        self.formal_params = formal_params

def logo_apply(proc, args):
    """Apply a Logo procedure to a list of arguments."""
    if proc.isprimitive:
        return proc.body(*args)
```

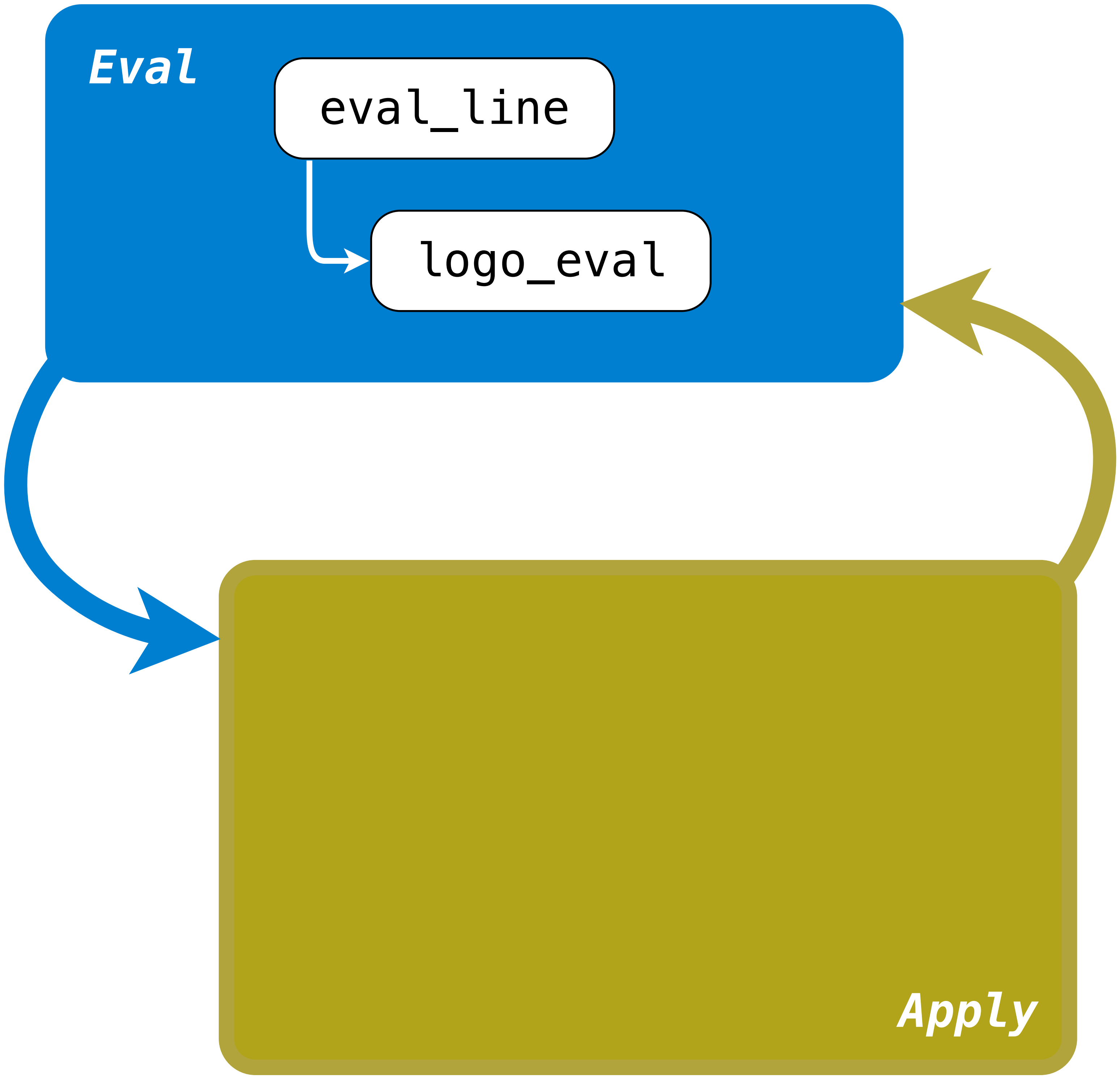
Procedures

```python
class Procedure():
    def __init__(self, name, arg_count, body, isprimitive=False,
                 needs_env=False, formal_params=None):
        self.name = name
        self.arg_count = arg_count
        self.body = body
        self.isprimitive = isprimitive
        self.needs_env = needs_env
        self.formal_params = formal_params

def logo_apply(proc, args):
    """Apply a Logo procedure to a list of arguments."""
    if proc.isprimitive:
        return proc.body(*args)
    else:
        """Apply a user-defined procedure"""
```

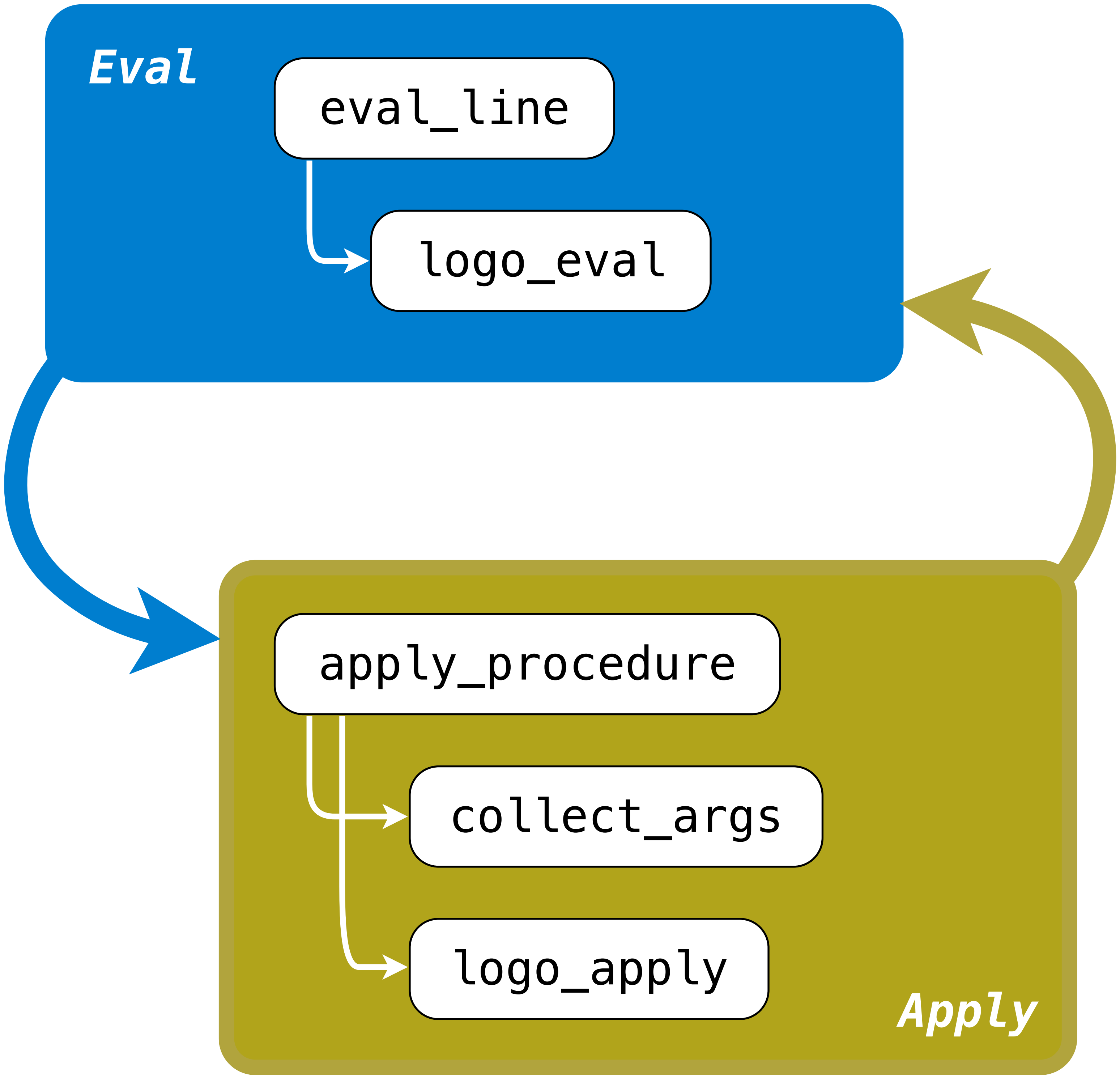
Logo Interpreter



Logo Interpreter



Logo Interpreter

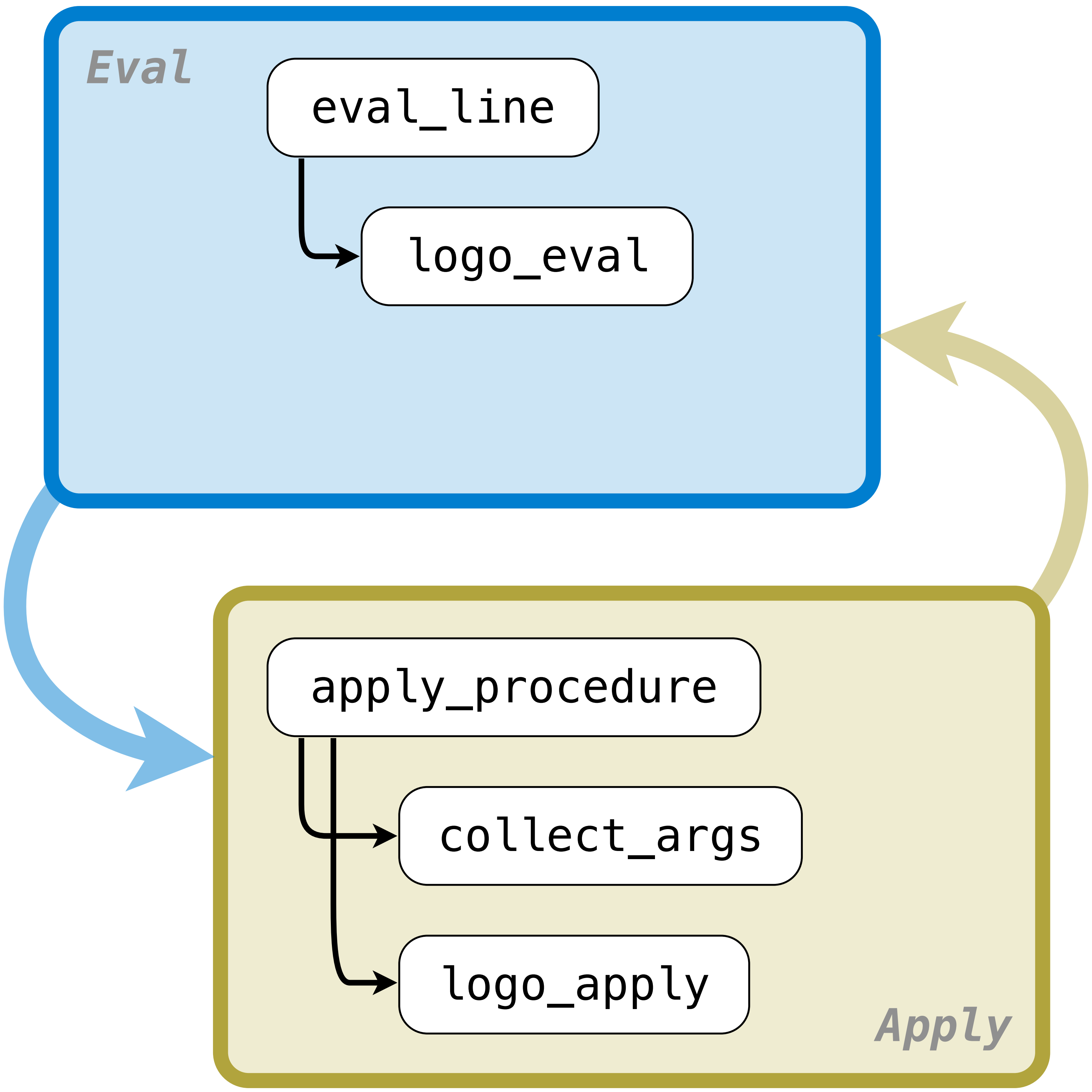


Eval/Apply in Lisp 1.5

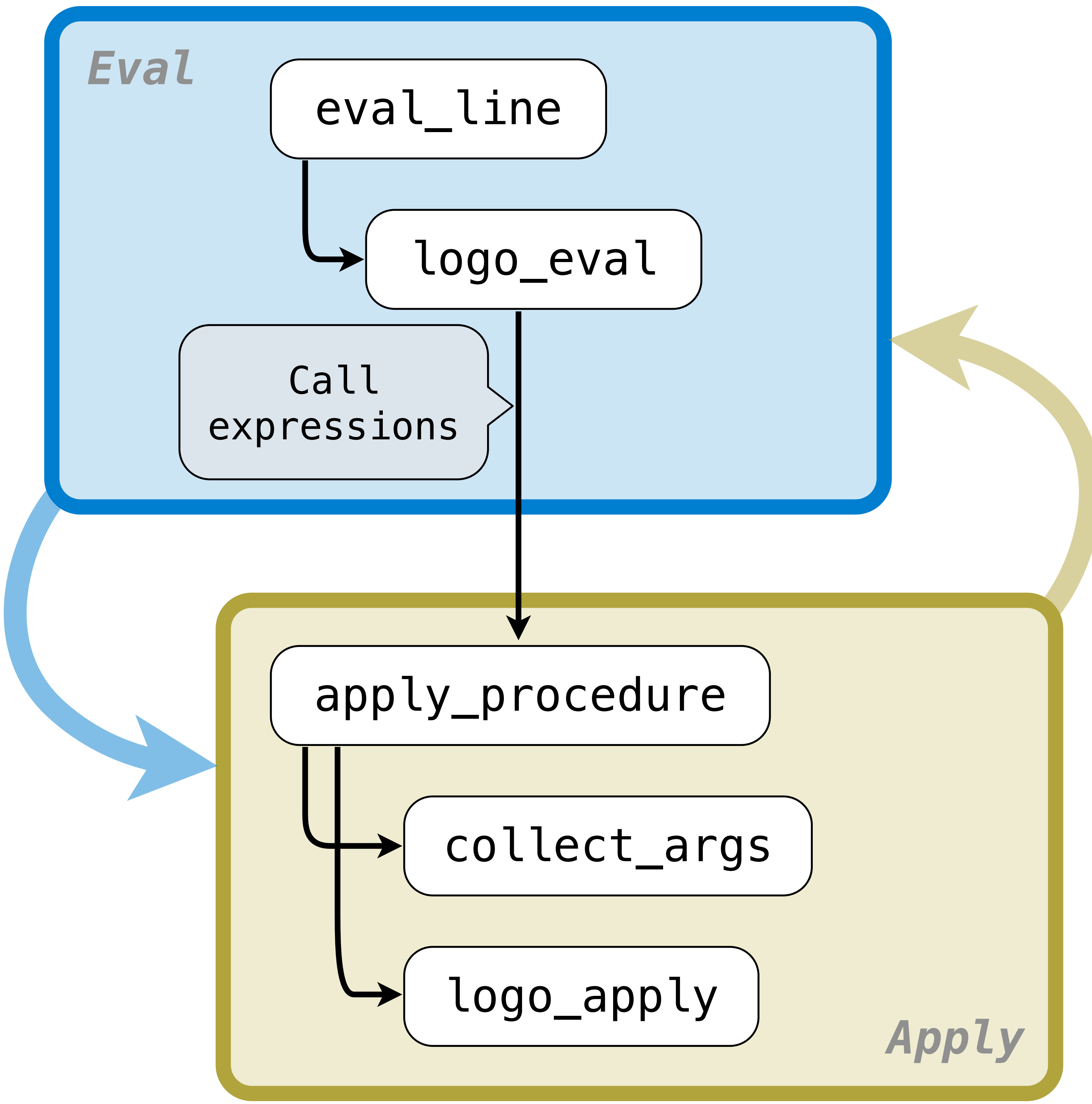
Eval/Apply in Lisp 1.5

```
apply[fn;x;a] =
     [atom[fn] → [eq[fn;CAR] → caar[x];
                  eq[fn;CDR] → cdar[x];
                  eq[fn;CONS] → cons[car[x];cadr[x]];
                  eq[fn;ATOM] → atom[car[x]];
                  eq[fn;EQ] → eq[car[x];cadr[x]];
                  T → apply[eval[fn;a];x;a]];
     eq[car[fn];LAMBDA] → eval[caddr[fn];pairlis[cadr[fn];x;a]];
     eq[car[fn];LABEL] → apply[caddr[fn];x;cons[cons[cadr[fn];
                                                   caddr[fn]];a]]]
eval[e;a] = [atom[e] → cdr[assoc[e;a]];
     atom[car[e]] →
               [eq[car[e],QUOTE] → cadr[e];
               eq[car[e];COND] → evcon[cdr[e];a];
               T → apply[car[e];evlis[cdr[e];a];a]];
     T → apply[car[e];evlis[cdr[e];a];a]]
```

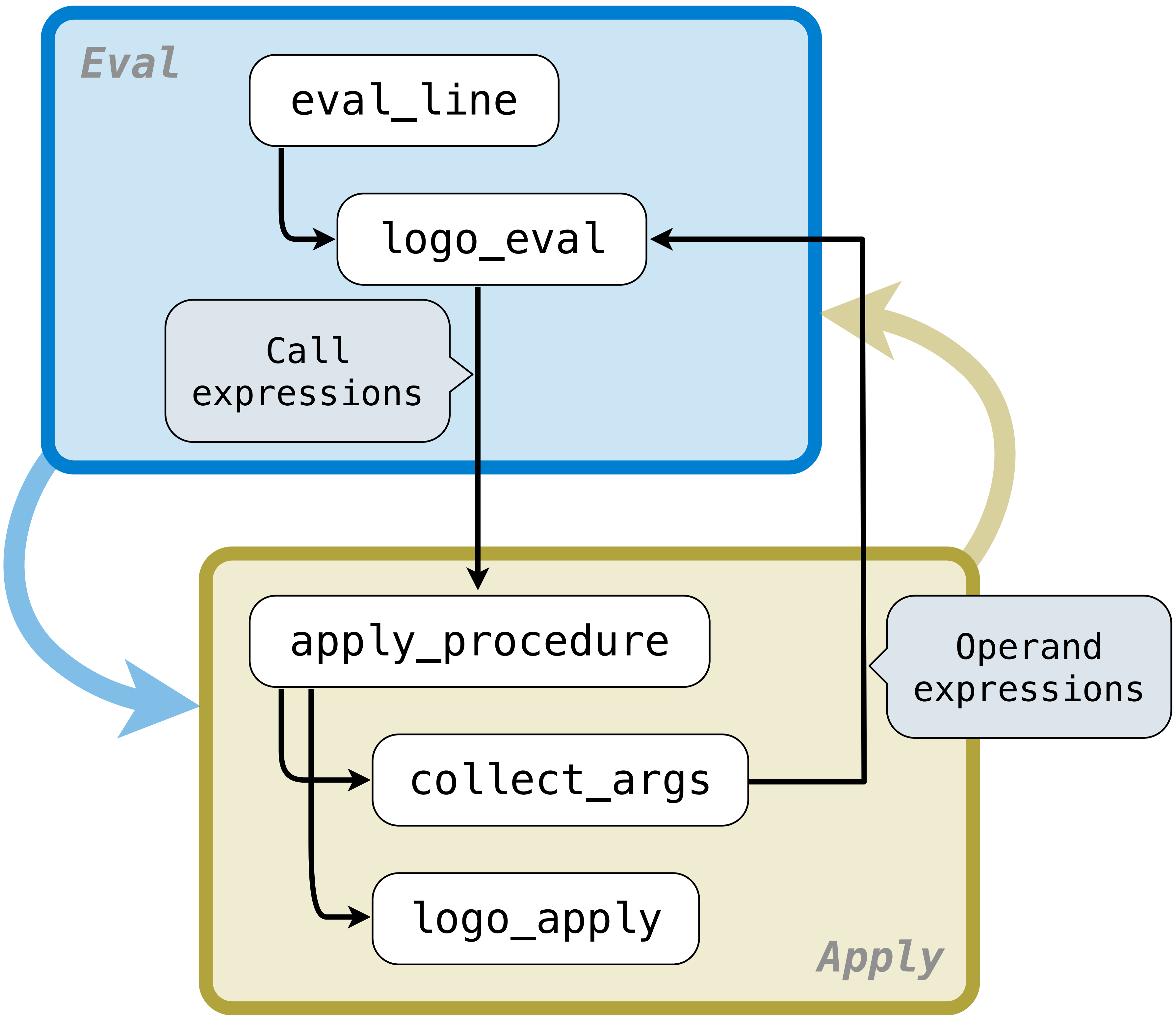
Eval/Apply in Logo



Eval/Apply in Logo



Eval/Apply in Logo



Eval/Apply in Logo