Lab 12

Scheme Lists, Interpreters

Mingxiao Wei
mingxiaowei@berkeley.edu

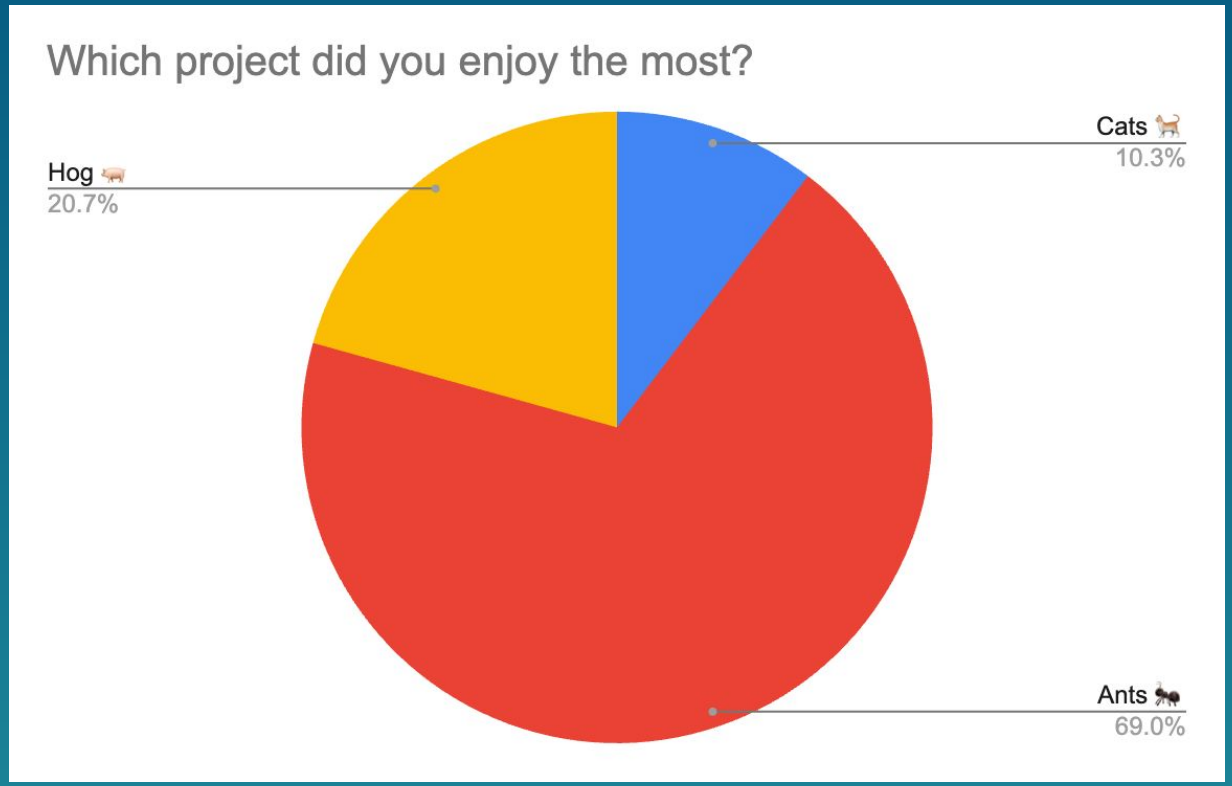
April 18, 2023

Apologize for the sudden switch of slides format… but I have to admit that google slides are much more powerful at producing visualizations than raw html files

Logistics

- Lab 12 due tomorrow 04/19
- Homework 09 due Thu 04/20
- Scheme is released! 👀
  - Checkpoint 1 (part 1) due this Fri 04/21
  - Checkpoint 2 (part 2 & 3) due next Tue 04/25
  - The entire project due next Fri 04/28
  - Submit everything by next Thu 04/27 for 1 extra credit!
  - Attempt the problems in order due to some dependency
  - Timing might be a lil tight but we do have a lot of OH/project parties!
  - Recommended to work with a partner - pls don't just split the work, but try to sit down and work on the project together!
- (optional) Scheme Contest - create cool art with Scheme!! 🧑‍🎨

From last time … 👀



From last time … 👀

“you recommended cs 164 today. Have you taken it? What other upper div CS classes have you enjoyed? :)”

- No I haven’t taken (and currently do not plan to take) 164 due to tight schedule :(
- But other classes I’ve taken include:
  - CS
    - lower divs: 61a/b/c, 70, eecs 16a/b
    - Upper divs: data 100 (useful data science), ee 120 (“Signals and systems” - linear algebra, convolution, Fourier transform, etc.), cs 160 (user interface), cs 170 (algorithms), cs 189 (machine learning)
  - MCB
    - All lower divs (math 1a/b as the math req)
    - Upper divs: mcb 102 (biochem), mcb 160 (cellular/molecular neurobio), mcb 161 (circuit/system neurobio), mcb 165 (neurobio of diseases)
  - If you are interested in learning more about these classes/anything related, feel free to talk to me after section or email me! You can also just ask in the attendance form - I’ll reply like this in the next section :)

Scheme Lists

Scheme List

- All Scheme lists are linked lists
- `(car lst)` → the first element of the list (similar to link.first)
- `(cdr lst)` → the rest of the list, either another list or `nil` (similar to link.rest)
- `nil` or `()` → the empty list in Scheme
- `(null? lst)` → checks if `lst` is empty
- `(length lst)` → length of `lst`

```
scm> (define lst '(1 2 3))
lst
scm> (car lst)
1
scm> (cdr lst)
(2 3)
scm> (null? (cdr (cdr (cdr lst))))
#t
scm> (car (cdr lst))
2
scm> (length lst)
3
```

Constructing a Scheme List

- 3 ways of constructing a Scheme list


```
scm> (cons 1 (cons 2 (cons 3 nil)))
(1 2 3)
scm> (list 1 2 3)
(1 2 3)
scm> ‘(1 2 3) ; or (quote (1 2 3))
(1 2 3)
```

Constructing a Scheme list - `cons`

`(cons first rest)`

- Similar to a linked list constructor
- `first` - the first value in the list
- `rest` - must be another Scheme list or `nil`
- Useful in recursion
  - Handle the first value in each call
  - The rest is handled by a recursive call

```
scm> (cons 1 nil)
(1)
scm> (cons 1 (cons 2 nil))
(1 2)
scm> (define a (cons 1 (cons ‘a nil)))
a
scm> a
(1 a)
scm> (cons 6 a)
(6 1 a)
```

Constructing a Scheme list - `list`

`(list elem_1 elem_2 ...)`

- Take in an arbitrary number of elements in the list, *evaluate* each of them, and return as a Scheme list
- Useful when we know the exact elements of a list

```
scm> (list 1)
(1)
scm> (list 1 2)
(1 2)
scm> (define a 6)
a
scm> a
6
scm> (list (- a 1) a (+ a 1))
(5 6 7)
```

Constructing a Scheme list - `quote`

`‘(...)` or `(quote ...)`

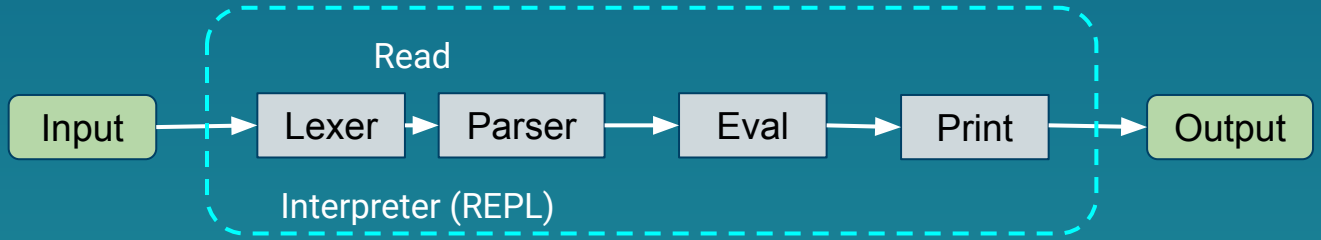
- Return the list as it is, <u>without any evaluation</u>
- Useful when we know what the list looks like in Scheme

```
scm> ‘(1)
(1)
scm> (quote (1 2))
(1 2)
scm> (define a 6)
a
scm> a
6
scm> (list 5 a 7)
(5 6 7)
scm> ‘(5 a 7) ; no evaluation
(5 a 7)
```

Interpreters

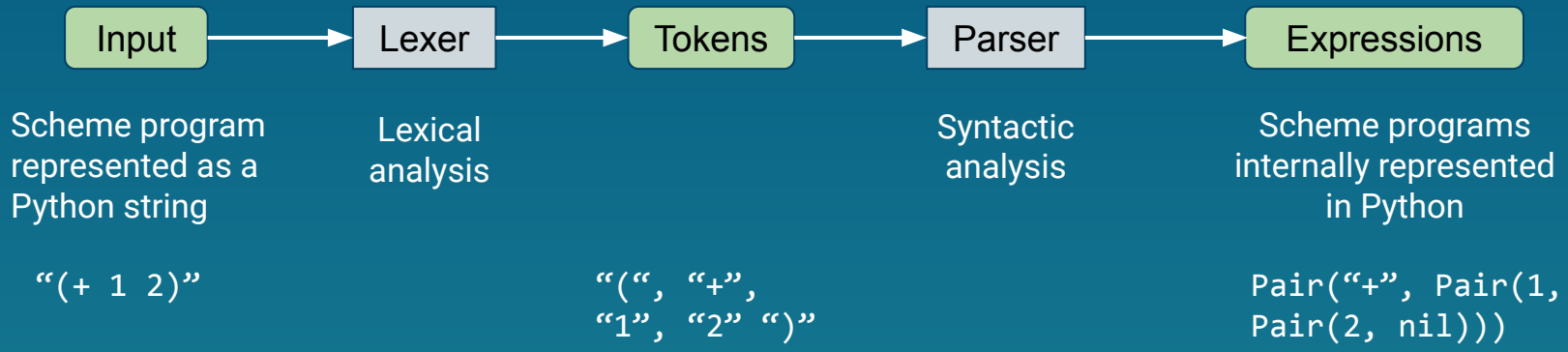
Interpreters

- **<u>Interpreter</u>** - a program that allows one to interact with the computer in a certain language
  - Think of it as a translator
  - Two languages at work
    - The language being interpreted (e.g., Scheme)
    - The language that performs the underlying interpretation (e.g., Python)
- Most interpreters use a REPL (Read-Eval-Print Loop)




```
scm> (+ 1 2)
3
```

Read stage


Input
Lexer
Tokens
Parser
Expressions
Scheme program represented as a Python string
Lexical analysis
Syntactic analysis
Scheme programs internally represented in Python
“(+ 1 2)”
“(“, “+”, “1”, “2” “)”
Pair(“+”, Pair(1, Pair(2, nil)))

Internal Representations

Expressions

How are scheme programs represented in the output of the read stage?

Scheme	Python
Numbers	`int` or `float` values
Symbols	String (`str`) values
Booleans (`#t`, `#f`)	`bool` values (`True`, `False`)
Combinations (anything that's not a primitive value - lists, call expressions, special forms)	`Pair` objects
`nil`	The `nil` object

No evaluation at the read stage - everything is a either primitive value (number, boolean, string) or a `Pair` object containing primitive values

The pair class

```
class Pair:
    """Represents the built-in pair data structure in Scheme."""
    def __init__(self, first, rest):
        self.first = first
        if not scheme_valid_cdrp(rest):
            raise SchemeError("cdr can only be a pair")
        self.rest = rest
    def map(self, fn):
        """Maps fn to every element in a list, returning a new Pair"""
        assert isinstance(self.rest, Pair) or self.rest is nil
        return Pair(fn(self.first), self.rest.map(fn))
```

Similar to a linked list - `first` is the value at the current node and `rest` is another `Pair` object or `nil` (empty Pair)

`rest` is not optional

To apply a one-argument function to every element in a Pair, do `pair_object.map(fn)`

The `nil` class/object

```
class nil:
    """Represents the special empty pair nil in Scheme."""
    def map(self, fn):
        return nil
    def __getitem__(self, i):
         raise IndexError('Index out of range')
    def __repr__(self):
        return 'nil'
```

```
nil = nil() # this hides the nil class *forever*
```

- `nil` represents the empty `Pair` - similar to `Link.empty`
- `nil` is an object - use `pair is nil` to check if a `Pair` object is empty

Convert a Scheme combination to a `Pair` object

- Each element in the Scheme combination corresponds to one node/element in the `Pair` object
- Length of Scheme combination = length of `Pair`
- Nested combination → nested `Pair`

“(+ 1 2)” → `Pair(“+”, Pair(1, Pair(2, nil)))`

“(+ 1 (* 2 3))” → `Pair(“+”, Pair(1, Pair(_____, nil)))`

Convert a Scheme combination to a `Pair` object

- Each element in the Scheme combination corresponds to one node/element in the `Pair` object
- Length of Scheme combination = length of `Pair`
- Nested combination → nested `Pair`

```
"(+ 1 2)"          →  Pair("+", Pair(1, Pair(2, nil)))

"(+ 1 (* 2 3))"    →  Pair("+", Pair(1, Pair(______, nil)))
```

↓

```
Pair("*", Pair(2, Pair(3, nil)))
```

Convert a Scheme combination to a `Pair` object

- Each element in the Scheme combination corresponds to one node/element in the `Pair` object
- Length of Scheme combination = length of `Pair`
- Nested combination → nested `Pair`

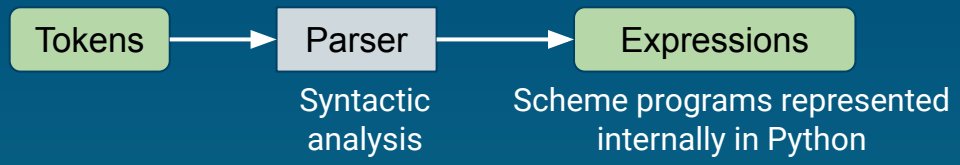
“(+ 1 2)” → `Pair(“+”, Pair(1, Pair(2, nil)))`

“(+ 1 (* 2 3))” → `Pair(“+”, Pair(1, Pair(Pair(“*”, Pair(2, Pair(3, nil))), nil)))`

↓

`Pair(“*”, Pair(2, Pair(3, nil)))`

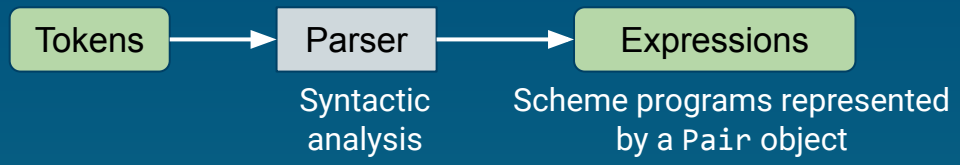
Syntactic analysis



- Identify the hierarchical structure of the program - may be nested
- Convert token to their internal representation in Python
  - Each `scheme_read` call consumes the input tokens for exactly one expression
- Recursive
  - Base case
    - primitive values (numbers, booleans, symbols)
    - Return the corresponding primitive value in Python
  - Recursive case
    - recursively call `scheme_read` to read the sub-expression and combine
    - Return the expression as a `Pair` object in Python

* In the Scheme project, this function is called `scheme_read`

Syntactic analysis



"(+ 1 (* 2 3))"

Tokenized by the lexer

"(", "+", "1", "(", "*", "2", "3", ")", ")"

Parser - syntactic analysis

Returns the expression

Pair("+", Pair(1, Pair(Pair("*", Pair(2, Pair(3, nil))), nil)))

scheme_read(["(", "+", "1", "(", "*", "2", "3", ")", ")"])

"+"

1

Pair("*", Pair(2, Pair(3, nil)))

scheme_read(["+"])

scheme_read(["1"])

scheme_read(["(", "*", "2", "3", ")"])

3

"*"

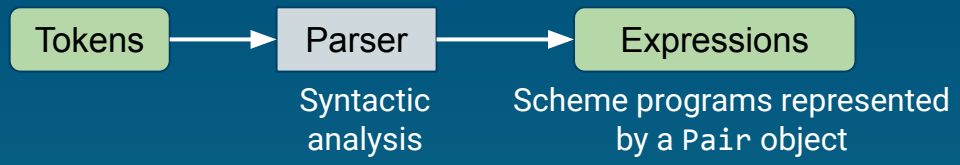
2

scheme_read(["*"])

scheme_read(["2"])

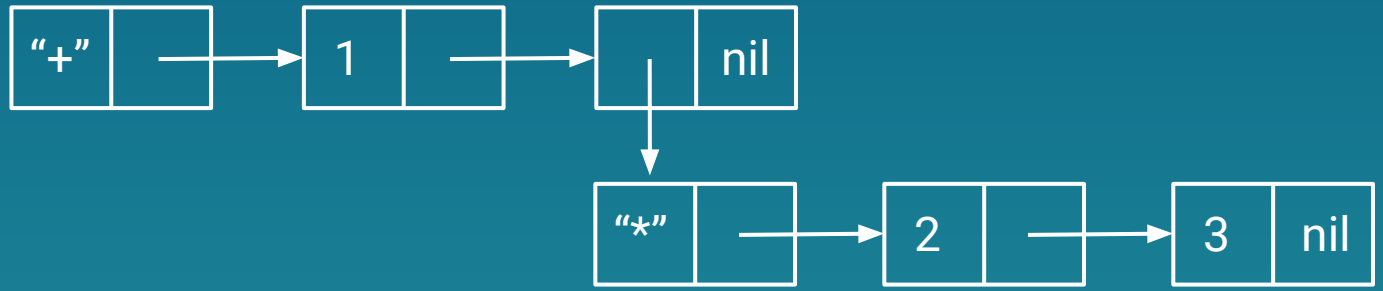
scheme_read(["3"])

Syntactic analysis

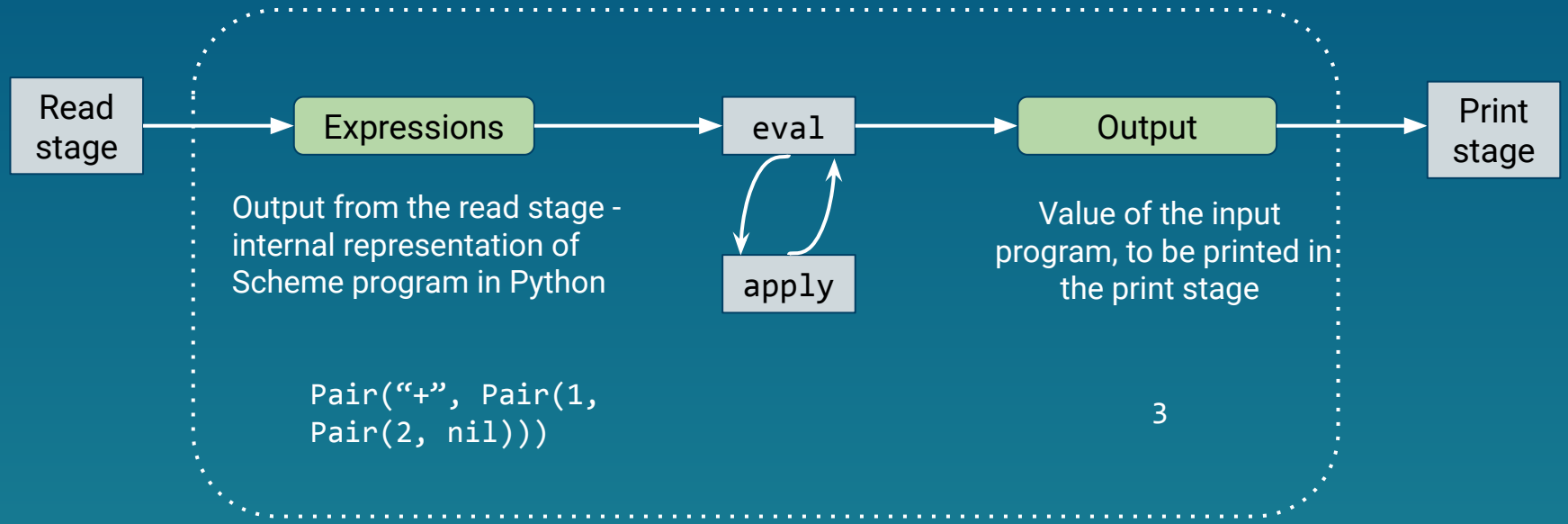



```
(+ 1 (* 2 3))
```

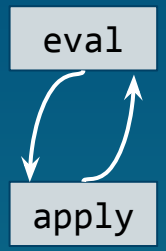
```
Pair(“+”, Pair(1, Pair(Pair(“*”, Pair(2, Pair(3, nil))), nil)))
```



Eval stage

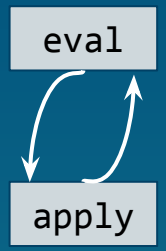

Read stage
Expressions
eval
Output
Print stage
Output from the read stage - internal representation of Scheme program in Python
apply
Value of the input program, to be printed in the print stage
Pair(“+”, Pair(1, Pair(2, nil)))
3

Eval/Apply



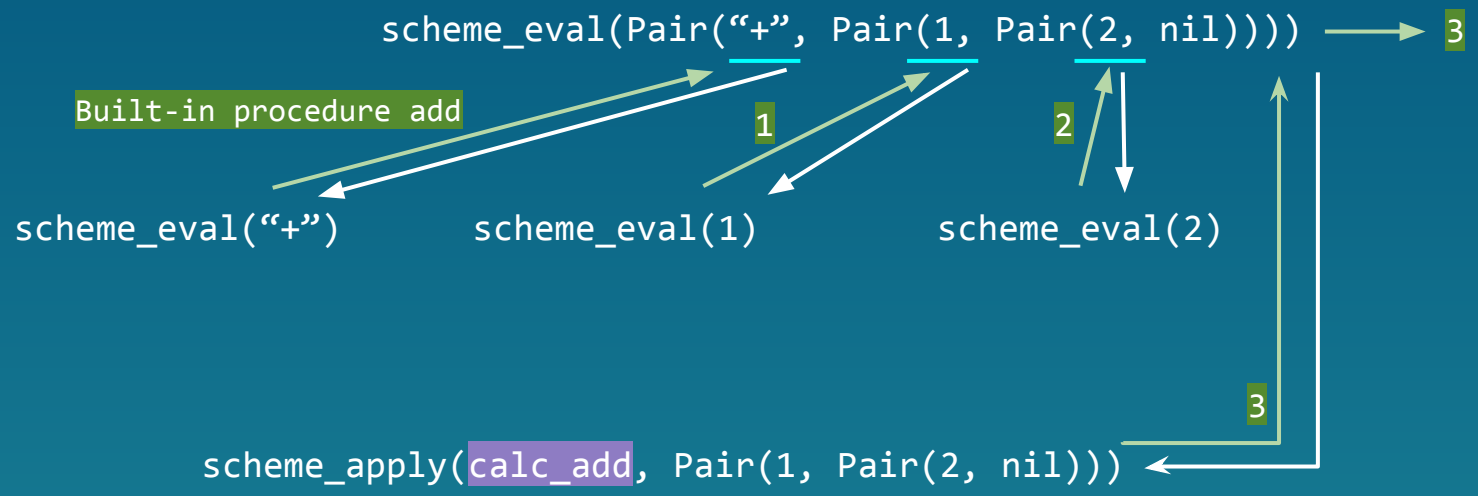
- `eval(exp)`
  - Takes in the output from the read stage as the input
  - Recursively evaluate the expression
    - Base case
      - Primitive values - return the value
      - Symbols - look up the variable name and return its value
    - Recursive case
      - Look at the first element in the expression
      - Special form → follow the corresponding evaluation rule
      - Call expression → recursively call `eval` on the operator and operands, then `apply` the operator to the operands

Eval/Apply



- `apply(op, args)`
  - Inputs
    - op - the evaluated operator, which is a function value
    - `args` - the evaluated operands as a `Pair` object
  - Apply the operator to its operands (recursively)
    - Base case
      - Built-in procedures
    - Recursive case
      - User-defined procedures - recursively call `eval` to evaluate the function body

Eval/Apply



The corresponding Python function for procedures like this should take in a Pair object as its argument, and return the desired result

Attendance 🤠

go.cs61a.org/mingxiao-att