Announcements

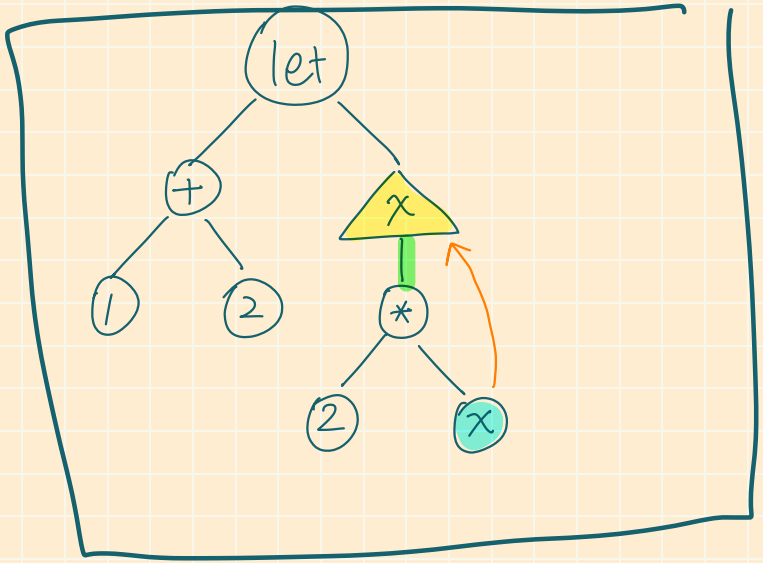
- HW 2 will be released later today, due in a week
- Same format as HW 1
- Quiz 1 will be held in class on July 15th
- No section tomorrow (happy 4th of July)!

Today's Plan

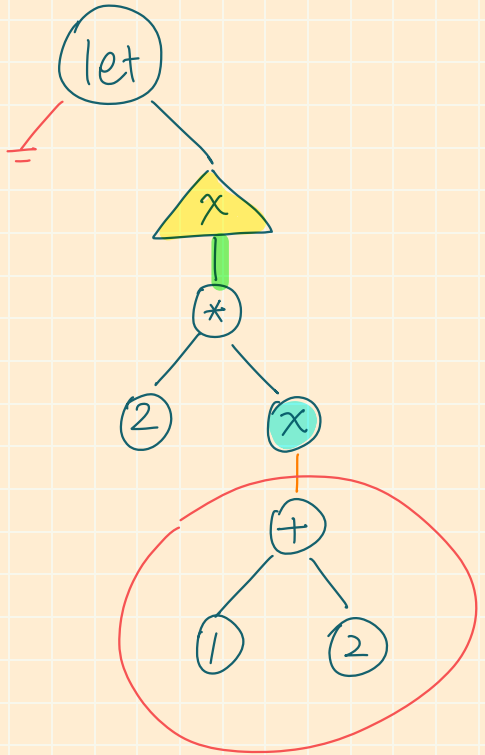
- 1 lecture about 45-60 mins
- Then we go over HW 1 together

A question I got yesterday

let x = 1+2 in 2*x

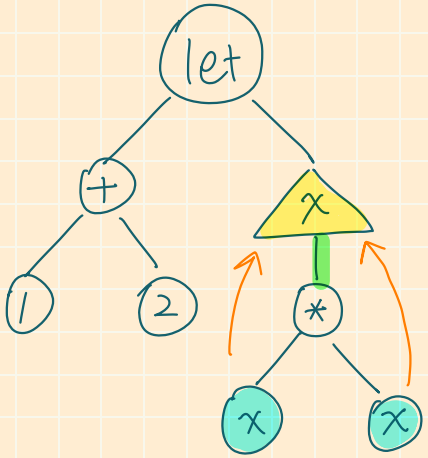


Why not?



Reason 1:
multiple references can point to the same declaration (sharing)

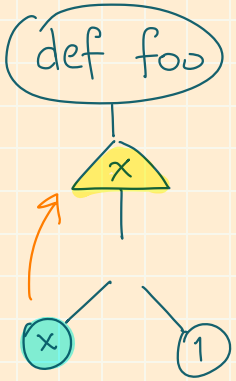
let x = 1+2 in x*x



Reason 2:
some language features only have the binder part (w/o the "=")

Functions:

```
def foo(x):
  return x + 1
```



Correction

How to know which declaration a reference points to?

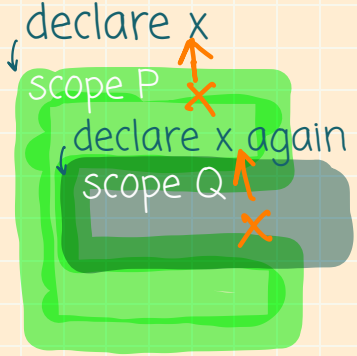
Every declaration of x is associated with a scope: ← Lexical scoping

- Scope = a subprogram or sub-expression
- Every reference to x inside the scope points to the declaration with which the scope is associated

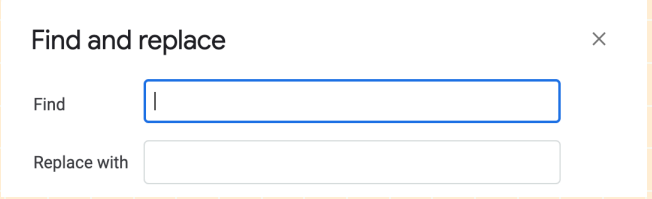
Caveat: scopes can be nested.

If you declare a variable x with scope P, and inside P, you declare variable x again with scope Q, then all references to x refers to the declaration associated with Q, not P.

↑ Shadowing

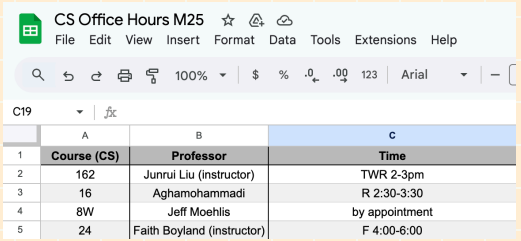


Substitution



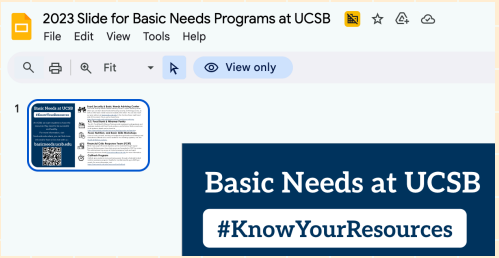
Google docs

Relation



Google sheets

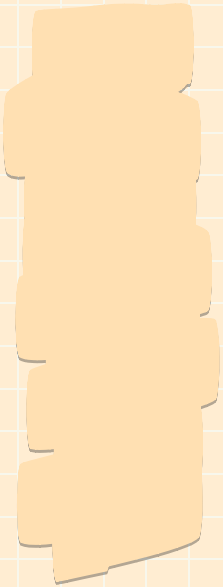
?



Google slides

Review

For each expr, identify every var as a declaration or a reference (free/bound)



```
(x + 1)

let y = x in y

let y = x in x

let x = y in x

let x = x in x
```

Review

Perform the following substitutions

Substitutions	Results
`[3/x] (x + 1)`	3+1
`[3/x] let y = x in y`	let y=3 in y
`[3/x] let y = x in x`	
`[3/x] let x = y in x`	let x=y in x
`[3/x] let x = x in x`	let x=3 in x

Review

Fill in the blank:

[r/x] e replaces all _free_ (free/bound) reference to x with _r_ in _e_

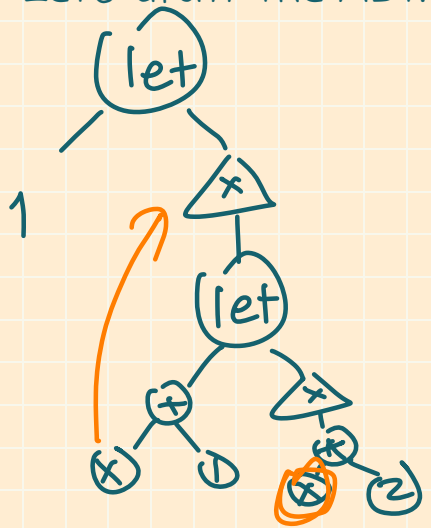
Why free?

Why does substitution replaces free references?

Consider

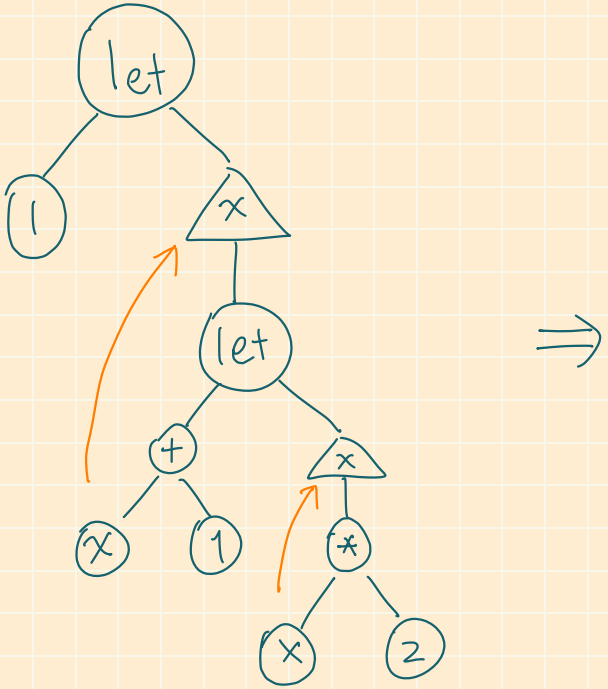
```
let x = 1 in
let x = x+1 in
x*2
```

Let's draw the ABT:

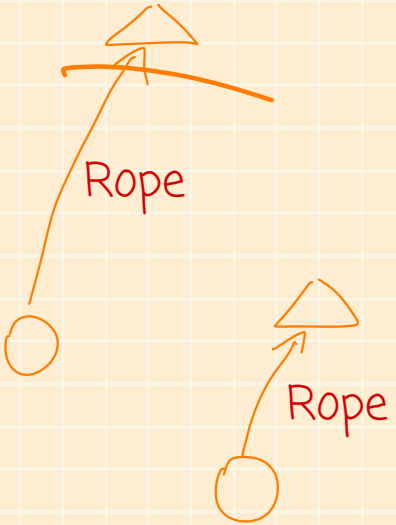


Q: if we look at the first binder x, how would you identify all references that point to it?

ABT = AST with backward edges pointing from reference to declaration.



Binding structure https://www.youtube.com/watch?v=1JpdVV-D6c14



Exercise: define a function **free_vars** that returns the set of all free variables (references) in an ABT

```
free_vars(e) =
  if e is Num:

  if e is Var(x):

  if e is Add(e1, e2):

  if e is Let(e1, x. e2):
```

Lambda Calculus

Variables + substitution = Turing complete

A bit of history

David Hilbert (1862-1943)

Entscheidungsproblem (decision problem)

Given any math theorem, is there an algorithm that outputs yes iff the theorem is true?

What is an algorithm?

What is an algorithm?

Alonzo Church

Lambda calculus
(1932)

Kurt Gödel

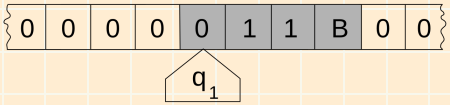
General recursion
(1933)

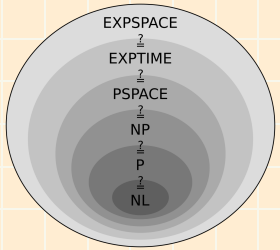
Alan Turing

Turing machines
(1936)

Church-Turing Thesis: every effectively computable function can be computed by a Turing machine.

Turing machine is to complexity theory





Lambda calculus is to programming language theory

lexical scoping

variables

`lambda x: x+1` `lambda expressions (c++11)`

Half-Life (1998) is to modern games

Ground breaking !

Extending our language with lambda calculus

```
expr ::= nat
       | e2 + e1
       | e1 * e2
       | let x = e1 in e2
       | x
       | λ x. e
       | e1 e2
```

λ-calc

`| λ x. e` ← Lambda function

`| e1 e2` ← Application

$(\lambda x.e)\ e_2$

Informal semantics:

- `λx. e` represents a (nameless) function with a single parameter x
- `(e1 e2)` means applying function e1 to argument e2


Always has been
if = λb.λx.λy. b x y
true = λx.λy. x
false = λx.λy. y
It's lambdas all the way down?!
0 = λb.λi. b
succ = λb.λi.λp.λr.
i p (r b i)

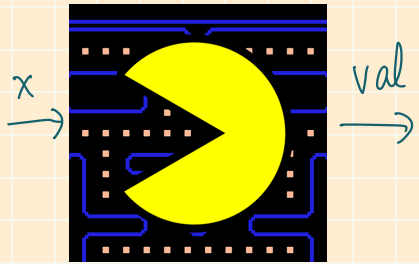
Examples:

$\lambda x.$ (x+1) body

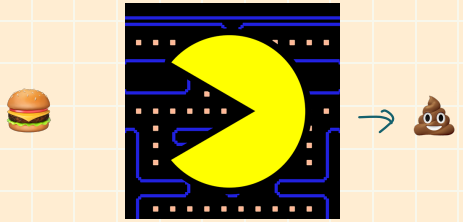
- A lambda function that increments its argument
- A lambda function that doubles its argument $\lambda x.\ x*2$
- Applying the increment function

$(\lambda x.\ x+1)\ 3 \Rightarrow 4$

Function =



Application = "eating" & "shitting"

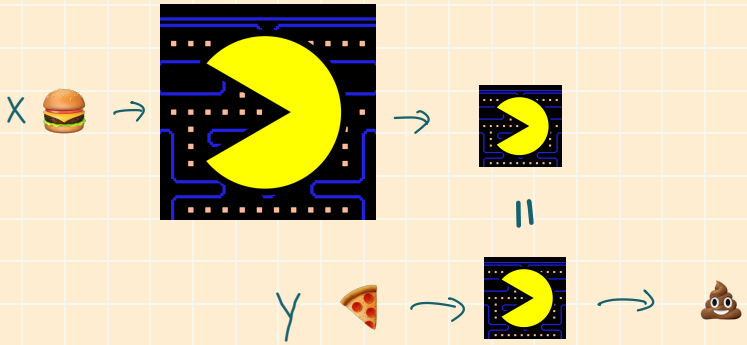


What about multi-argument functions?

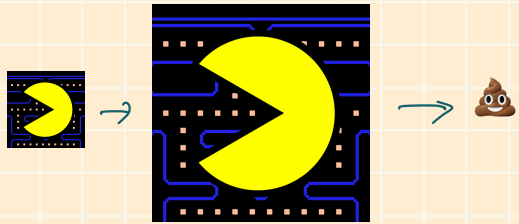
A lambda function that takes two arguments and adds them

$$\lambda x.(\lambda y.\ x+y\ )$$

Higher-order functions: returning another function

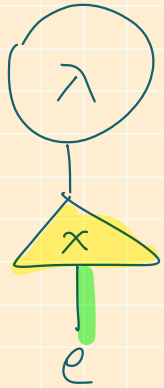


Higher-order functions: eating another function

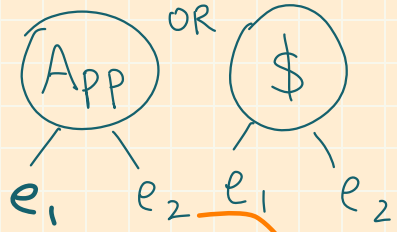


ABTs

`λx. e` declares x with scope e

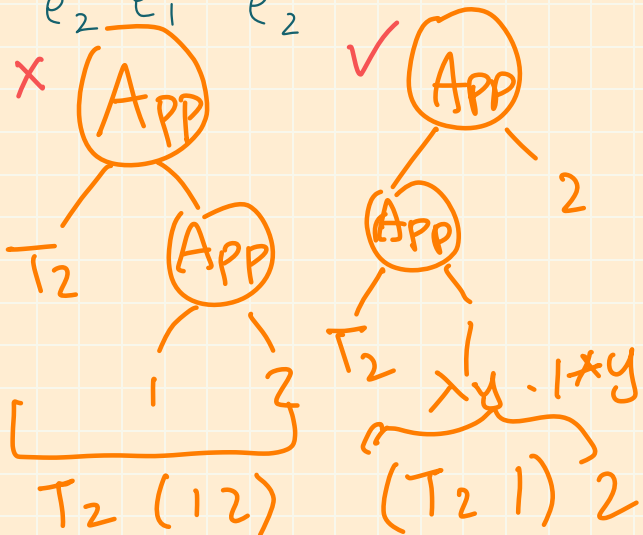


`e1 e2` No binding



Exercises: draw the ABTs for:

1. `λx. x+1`
2. `λx. λy. x*y`
3. `((λx. λy. x*y) 1 2)` tricky!
4. `let x = λx. x*2 in x 1`



Operational Semantics

`expr ⇒ value`

```
expr ::= nat
       | e2 + e1
       | e1 * e2
       | let x = e1 in e2
       | x
       | λ x. e
       | e1 e2

value := nat
       | λx. e
```

$$\frac{}{\lambda x.e \Rightarrow \lambda x.e}\ \lambda$$

$$\frac{e_1 = \lambda x.e \quad e_1 \Rightarrow \lambda x.e \quad e_2 \Rightarrow v \quad [v/x]e = e' \quad e' \Rightarrow v'}{e_1\ e_2 \Rightarrow v'}\ App$$

Examples:

$$\frac{}{n \Rightarrow n}\ \text{Nat} \qquad \frac{e_1 \Rightarrow n_1 \quad e_2 \Rightarrow n_2 \quad (n_1 + n_2 = n_3)}{e_1 + e_2 \Rightarrow n_3}\ \text{Add}$$

$$\frac{}{\lambda x.e \Rightarrow \lambda x.e}\ \lambda \qquad \frac{e_1 \Rightarrow \lambda x.e \quad e_2 \Rightarrow v \quad [v/x]e = e' \quad e' \Rightarrow v'}{e_1\ e_2 \Rightarrow v'}\ \text{App}$$

$$\frac{\dfrac{}{\lambda x.\, 2+x \Rightarrow \lambda x.\, 2+x}\ \lambda \qquad \dfrac{\dfrac{}{1 \Rightarrow 1}\ \text{Nat} \quad \dfrac{}{2 \Rightarrow 2}\ \text{Nat} \quad (1+2=3)}{1+2 \Rightarrow 3}\ \text{Add} \qquad \dfrac{\dfrac{}{2 \Rightarrow 2}\ \text{Nat} \quad \dfrac{}{3 \Rightarrow 3}\ \text{Nat} \quad (2+3=5)}{2+3 \Rightarrow 5}\ \text{Add} \quad ([3/x]\,2+x = 2+3)}{(\lambda x.\, 2+x)\ (1+2) \Rightarrow 5}\ \text{App}$$

Examples:

$$\frac{}{n \Rightarrow n}\ \text{Nat} \qquad \frac{e_1 \Rightarrow n_1 \quad e_2 \Rightarrow n_2 \quad (n_1 + n_2 = n_3)}{e_1 + e_2 \Rightarrow n_3}\ \text{Add}$$

$$\frac{}{\lambda x.e \Rightarrow \lambda x.e}\ \lambda \qquad \frac{e_1 \Rightarrow \lambda x.e \quad e_2 \Rightarrow v \quad [v/x]e = e' \quad e' \Rightarrow v'}{e_1\ e_2 \Rightarrow v'}\ \text{App}$$

HW

$$\frac{}{((\lambda x.\ \lambda y.\ x+y)\ 1)\ 2}$$

Problem 2-A (✍ written, 2 pts)

Formalize the abstract syntax of propositional logic propositions using context-free grammar (CFG). Your grammar should support `~` (unary negation), `/\` (binary conjunction), `\/` (binary or), `->` (binary implication), `<->` (iff), along with boolean constants (**True** and **False**) and propositional variables ($P, Q, R, \ldots$).

prop ::= ~ prop
| prop ∧ prop
| prop ∨ prop
| prop ⇒ prop
| prop ↔ prop
| True | False | Var / P, Q, R, ... (or)

Problem 2-B (✍ written, 2 pts)

Use Python's `dataclass` to represent the abstract syntax tree (AST) of propositions defined by your grammar. Name the abstract class `Prop`, and include the class definitions in your PDF:

```
@dataclass(frozen=True)
class Prop:
    pass

@dataclass(frozen=True)
class <Class1>(Prop):
    """Explanation of how you obtained Class1 from a CFG production rule."""
    # fields

@dataclass(frozen=True)
class <Class2>(Prop):
    """Explanation of how you obtained Class2 from a CFG production rule."""
    # fields

...
```

For each class, informally explain how you translated the CFG production rule into that class definition using 1 sentence in the docstring (enclosed in `"""`).

True False

Soln 1

```
class Const(Prop):
    value: bool
```

or

Soln 2

```
class CTrue(Prop):
    pass

class CFalse(Prop):
    pass
```

Problem 2-C (✍ written, 2 pts)

Pretend you're the parser. For each of the following proposition:

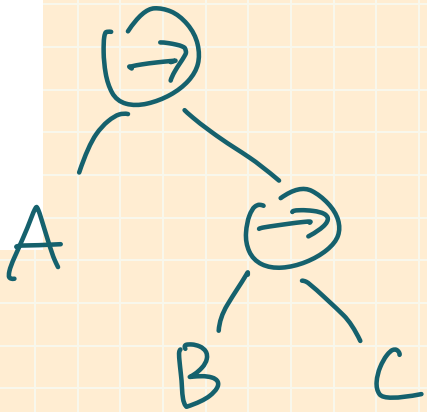
1. draw the corresponding AST corresponding to the formula in concrete syntax
2. write down a Python object (of class `Prop`) that represents the same AST. The object should be constructed using a series of constructor calls for your `<Class1>`, `<Class2>`, etc. that you defined in Problem 2-B.

Propositions:

- `A -> B -> C`
- `A \/ B \/ C`
- `A /\ B \/ C`
- `-A \/ -B <-> -(A /\ B)`

A op B op C ⇒ A op (B op C)

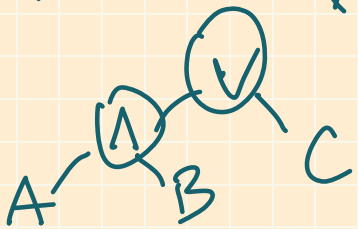
A → (B → C)



Use the following precedence and associativity rules to resolve any ambiguities:

- `-` has the highest precedence and is non-associative (since it's unary).
- `/\` has the second highest precedence and is left associative.
- `\/` has the third highest precedence and is left associative.
- `->` and `<->` have the lowest precedence and are right associative.

∧ > ∨ (A ∧ B) ∨ C



Formalize the meaning of logical formulas by inductively defining the judgment

$$\alpha \vdash P \Rightarrow b$$

arity = 3

mapping prop boolean value

using inference rules.

- P is a proposition
- b is a boolean value (**True** or **False**)
- α is an assignment (a partial function) from propositional variables to boolean values. You can write α using the following grammar:

$$\alpha ::= \varnothing \mid \alpha, x \mapsto b$$

$$\frac{\alpha \vdash P \Rightarrow \text{false}}{\alpha \vdash P \wedge Q \Rightarrow \text{false}} \wedge fL \qquad \frac{\alpha \vdash Q \Rightarrow \text{false}}{\alpha \vdash P \wedge Q \Rightarrow \text{false}} \wedge fR$$

$$\frac{\alpha \vdash P \Rightarrow \text{true} \quad \alpha \vdash Q \Rightarrow \text{true}}{\alpha \vdash P \wedge Q \Rightarrow \text{true}} \wedge t$$

α : as a function (partial) from vars to boolean values

1. $$\frac{x \in \mathrm{dom}(\alpha) \quad \alpha(x) = b}{\alpha \vdash x \Rightarrow b}\ \text{Var}$$

2. α as a set of pairs $\{ (x, t); (y, f) \}$

$$\frac{(x, b) \in \alpha}{\alpha \vdash x \Rightarrow b}\ \text{Var}$$

3. $\alpha ::= \bullet \mid \alpha, x \mapsto b$

$$\frac{}{\alpha, x \mapsto b \vdash x \Rightarrow b}\ \text{Found} \qquad \frac{(y \neq x) \quad \alpha \vdash x \Rightarrow b}{\alpha, y \mapsto b \vdash x \Rightarrow b}\ \text{Lookup}$$

$$\frac{}{\alpha \vdash \text{True} \Rightarrow \text{True}} \qquad \frac{}{\alpha \vdash \text{False} \Rightarrow \text{False}}$$

Problem 3-A (✍ written, 2 pts)

The abstract syntax you designed for propositional logic in Problem 2 can be considered a surface syntax, since many operators can be expressed using a small set of primitives. Your task is to design a core syntax for propositional logic that has no more than 4 cases:

```
prop ::= True | False
       | Var
       | <op> (operator)
```