;

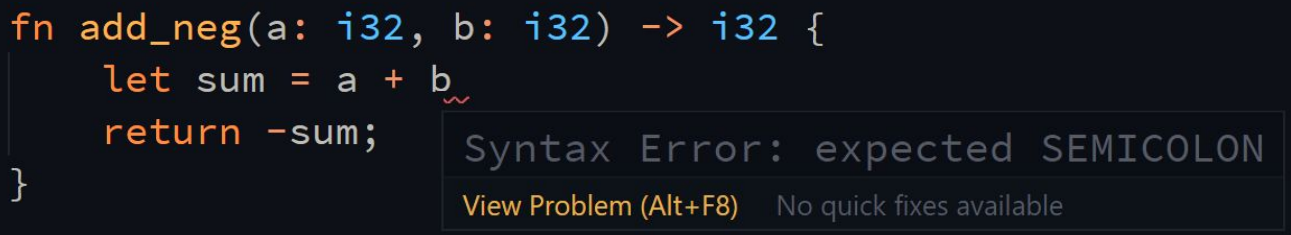
# why use semicolons?

```
a += 1; b += 2
```

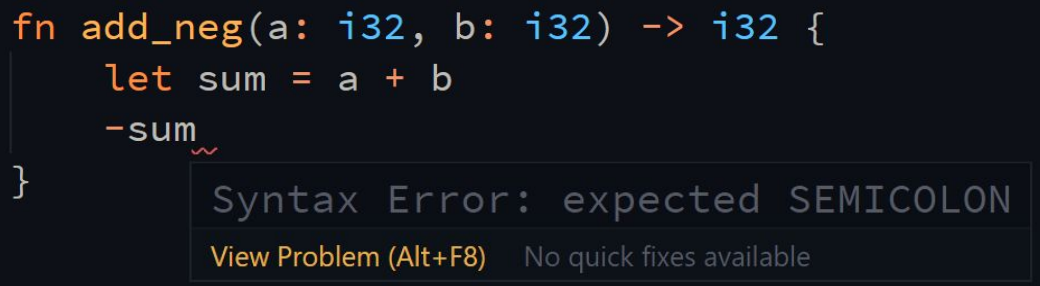
```rust
fn fib(n: i32) -> i32 {
    let mut a = 0;
    let mut b = 1;
    for i in 0..n {
        let c = a + b;
        a = b;
        b = c;
    }
    return a;
}
```

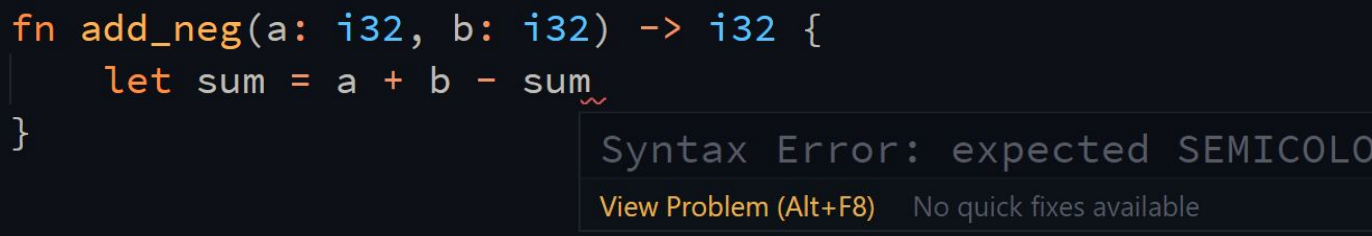
```
1; for i in 0..n { let c = a + b; a = b; b = c; } return
```

```rust
fn add_neg(a: i32, b: i32) -> i32 {
    let sum = a + b
    return -sum;
}
```


fn add_neg(a: i32, b: i32) -> i32 {
let sum = a + b
return -sum;
}
Syntax Error: expected SEMICOLON
View Problem (Alt+F8)
No quick fixes available

```rust
fn add_neg(a: i32, b: i32) -> i32 {
    let sum = a + b
    -sum
}
```


fn add_neg(a: i32, b: i32) -> i32 {
let sum = a + b
-sum
}
Syntax Error: expected SEMICOLON
View Problem (Alt+F8)
No quick fixes available


fn add_neg(a: i32, b: i32) -> i32 {
let sum = a + b - sum
}
Syntax Error: expected SEMICOLO
View Problem (Alt+F8)
No quick fixes available

```
let sum = a + b
*result = sum
```

```
let sum = a + b * result = sum
```

```
let x = foo
(a, b) = (b, a)
```

```
let x = foo(a, b) = (b, a)
```


Lua

```
block ::= {stat} [retstat]

stat ::=  ‘;’ |
         varlist ‘=’ explist |
         functioncall |
         label |
         break |
         goto Name |
         do block end |
         while exp do block end |
         repeat block until exp |
         if exp then block {elseif exp then block} [else block] end |
         for Name ‘=’ exp ‘,’ exp [‘,’ exp] do block end |
         for namelist in explist do block end |
         function funcname funcbody |
         local function Name funcbody |
         local attnamelist [‘=’ explist]
```

```
local x = foo
(a, b) = (b, a)
```

```
local x = foo
a, b = b, a
```

```lua
function add_neg(a, b)
    local sum = a + b;  -- optional semicolon
    -sum
end


function add_neg(a, b)
    local sum = a + b
    return -sum
end
```

```python
def foo():
    print("hello, weirdo")

    total = 0
    for i in range(5):
        print(i)
        total += (
              4*i*i + 3*i + 7
            + 8*i*i - 3*i)

    return total
```


GO

```go
package main

import "fmt"

func main() {

    fmt.Println("go" + "lang")

    fmt.Println("1+1 =", 1+1)
    fmt.Println("7.0/3.0 =", 7.0/3.0)

    fmt.Println(true && false)
    fmt.Println(true || false)
    fmt.Println(!true)
}
```

```go
package main

import "fmt"

func main() {

        fmt.Println("go" + "lang")

fmt.Println("1+1 =", 1+1)
            fmt.Println("7.0/3.0 =", 7.0/3.0)

    fmt.Println(true &&
    false        )

    fmt.Println(
        true || false)

    fmt.
    Println(!true)
}
```

```go
package main

import "fmt"

func main() {

        fmt.Println("go" + "lang")

fmt.Println("1+1 =", 1+1)
            fmt.Println("7.0/3.0 =", 7.0/3.0)

    fmt.Println(true &&
    false        )

    fmt.Println(
        true || false)

    fmt.
    Println(!true)
}
```

semicolon insertion:

- after a line's final token.
- if that token is one of:
  - identifier
  - literal
  - **break**
  - **continue**
  - **fallthrough**
  - **return**
  - ++ -- ) ] }

```
sum := first_part
    + 3*second_part
    + 8*last_part

GetUserName()
    .capitalize()
    .replaceAll("-")
```

```
sum := first_part
    + 3*second_part
    + 8*last_part

GetUserName()
    .capitalize()
    .replaceAll("-")
```

semicolon insertion:

- after a line's final token.
- if token can end an expression.
- *but the next token does not indicate that the statement continues.*

```rust
pub fn semi_colon_after(&self) -> bool {
    use TokenData::*;
    match self {
        // anything that can mark the end
        // of an expression.
        Ident (_) | Number (_) | Bool(_) | Nil | QuotedString(_) |
        RParen | RBracket | RCurly |
        KwBreak | KwContinue | KwReturn |
        KwEnv |
        KwEnd
        => true,

        _ => false,
    }
}
```

```rust
pub fn semi_colon_before(&self) -> bool {
    use TokenData::*;
    match self {
        // unless the next token indicates
        // that the expression may continue.
        RParen |
        RBracket |
        RCurly |
        Dot | Comma | Colon | SemiColon |
        KwEnd |
        KwElif | KwElse |
        KwIn |
        KwAnd | KwOr |
        OpPlus | OpPlusEq |
        OpMinus | OpMinusEq |
        OpStar | OpStarEq |
        OpSlash | OpSlashEq |
        OpSlashSlash | OpSlashSlashEq |
        FatArrow | ColonEq |
        OpEq | OpEqEq | OpNe | OpLe | OpLt | OpGe | OpGt |
        OpQ | OpQDot | OpQQ | OpQQEq |
        Error
        => false,

        _ => true,
    }
}
```

```
let sum = a + b
*result = sum
```

```
let sum = a + b * result = sum
```

```
fn add_neg(a, b)
    let sum = a + b
    return -sum
end
```

```
let neighbors =
      board_get(board, w, h, x - 1, y - 1)
    + board_get(board, w, h, x    , y - 1)
    + board_get(board, w, h, x + 1, y - 1)
    + board_get(board, w, h, x - 1, y    )
    + board_get(board, w, h, x + 1, y    )
    + board_get(board, w, h, x - 1, y + 1)
    + board_get(board, w, h, x    , y + 1)
    + board_get(board, w, h, x + 1, y + 1)

let cell = board[y*w + x]
if neighbors == 3:
    new_board[y*w + x] = 1
elif cell == 1 and neighbors == 2:
    new_board[y*w + x] = 1
end
```