CSE 231: Assignments and Blocks

Ranjit Jhala *April 21, 2026*

Assignments and Blocks

Next, we add **imperative** features, starting with

- **assignments** that let us *update* values of variables,
- **blocks** that let us *sequence* multiple expressions.

These are a stepping stone towards then adding **loops**.

Concrete Syntax

```
<expr> := ...
        | (set! <ident> <expr>)   -- update variable
        | (block <expr>+)         -- sequence exprs
```

QUIZ: Abstract Syntax

First, lets fill in the cases for `set!` and `block`

```
enum Expr {
  // ...


}
```

QUIZ: Semantics of `set!` and `block`

Program *Result*

```
(let (x 5)
  (set! x 10)
)
```

```
(let (x 10)
  (let (y (set! x (+ x 5)))
    (+ x y))
```

```
(let (x 5)
  (block
    (set! x (+ x 100))
    x
  )
)
```

QUIZ: Evaluator

Lets fill in the cases for the `eval` function.

How to
- *Update* a variable?
- *Sequence* expressions?

```
fn eval(expr: &Expr, env: &Env) -> i64 {
  match expr {
    // ...
    Expr::Set(x, e) => {

      ____________________________________________

      ____________________________________________

      ____________________________________________
    }
    Expr::Block() => {

      ____________________________________________

      ____________________________________________

      ____________________________________________

      ____________________________________________

      ____________________________________________
    }
  }
}
```

QUIZ: Assembly for `set!` and `block`

Complete the assembly code for

Program

```
(let (x 10)
  (let (y (set! x (+ x 1)))
    x)
```

Assembly

```
mov rax, 20
mov [rbp - 8.1], rax
mov rax, [rbp - 8.1]
add rax, 2
_____________________
_____________________
_____________________
```

```
(let (x 10)
  (block
    (set! x (+ x 1))
    x
  )
)
```

```
mov rax, 20
mov [rbp - 8.1], rax
mov rax, [rbp - 8.1]
add rax, 2
_____________________
_____________________
_____________________
```

QUIZ: Strategy for `set!` and `block`

```
(set! x e)
```

```
(block
  e0
  e1
  e2
)
```

Compilation Code

Lets fill in the cases for `compile_expr` for `set!` and `block`

```
fn compile_expr(expr: &Expr, env: &Env) -> String {
  match expr {
    // other cases ...
    Expr::Set(x, e) => {









    }

    Expr::Block(es) => {









    }
}
```

CSE 231: Loops

Ranjit Jhala *April 21, 2026*

Loops

Now lets add constructs for *iteration*, specifically

- **loop** *repeatedly* evaluates an expression, and
- **break** which gives us a way to *stop* iteration.

Concrete Syntax

```
<expr> := ...
        | (loop <expr>)
        | (break <expr>)
```

Idea

- `loop e` evaluates `e` repeatedly *until* it hits
- `break e` which exits returning the value of `e`

QUIZ: Semantics of `loop` and `break`

Program *Result*

```
(let (i 0)
  (loop
    (if (= i input)
      (break i)
      (set! i (+ i 1)))))
```

```
(let (res 0)
  (let (i 0)
    (loop
      (if (= i input)
          (break res)
          (block
             (set! res (+ res i))
             (set! i (+ i 1)))))))
```

Evaluator

Lets extend the `eval` function for `loop` and `break`.

Problem How to `break` out of a `loop` during evaluation.

Where does the value of `break` *go*? What can we *do* with it?

`loop` and `break` via "exceptions"

To evaluate `loop e`

- "Try" to evaluate `e` (and recursing to iterate)
- "Throw" an exception if you hit a `break`
- "Catch" the exception and return the value of `break`

Exceptions

We can define a type of **exceptions** to "throw" and "catch" during evaluation.

```
pub enum Exn {
  Break(Val),
  UnboundVar(String),
  TypeError(String),
}
```

Results

The `Result` type lets us represent computations that either

- succeed with a value `T` or
- fail with an error `E`.

```
enum Result<T, E> {
  Ok(T),
  Err(E),
}
```

Evaluation: Returns an `Result<Val, Exn>`

```
fn eval(expr: &Expr, env: &Env) -> Result<Val, Exn> {
 match expr {
  // ...
  Expr::Loop(body) => {
    match eval(body, env) {
      Ok(_) => eval(e, env),       // continue!
      Err(Err::Break(v)) => Ok(v), // exit!
      Err(e) => Err(e),            // other error!
    }
  }
  Expr::Break(e) => {
    match eval(e, env) {
      Ok(v) => Err(Exn::Break(v)), // throw break exn
      Err(e) => Err(e),            // other error!
    }
  }
}
```

Note we have to update all the other cases using `?` to propagate errors. See this link for more details on how to use `Result` in Rust: https://doc.rust-lang.org/book/ch09-02-recoverable-errors-with-result.html

QUIZ: Assembly for `loop` and `break`

Lets *complete* the assembly code for

Program

```
(let (i 0)
(loop
  (if (= i 5)
    (break 99)
    (set! i (+ i 1)))))
```

Assembly

```
our_code_starts_here:
  mov rax, 0
  mov [rbp - 8*1], rax

_____________________

  ; <(= i 5)>
  cmp rax, 1
  je if_false_1

if_true_1:

_____________________

_____________________

_____________________

if_false_1:
  ; <set! i (+ i 1)>

if_exit_1:

_____________________

_____________________
```

What extra *control-flow labels* do we need?

QUIZ: Strategy for `loop` and `break`

```
(loop e)
```

```
(break e)
```

Compilation Code

What extra information does `compile_expr` need to track?

```
fn compile_expr(expr: &Expr, env: &Env,              )
   -> String
{
  match expr {
    // other cases ...
    Expr::Loop(body) => {









    }

    Expr::Break(e) => {









    }
}
```