

# Compiler Design

## INF-400

Burak Arslan  
ext-inf400@burakarslan.com

Galatasaray Üniversitesi

Lecture III, Part II  
2026-10-02

## Course website

[burakarslan.com/inf400](http://burakarslan.com/inf400)

While the parsing work is going on, we have a lot to figure out about the details of our programming language.

## Karpuz

Language Definition

# Karpuz

## Laws of Karpuz

Note the following notation:

► → This is a law of Karpuz ←

It's one of the perks of being a language designer – we get to make  
our own laws !

Here is a variable declaration that *could* work.

```
id;
```

Here is a variable declaration that *could* work.

```
id;
```

But it's pointless so we don't allow it.

- ▶ → This is a compile-time error ←
- ▶ Variable 'id' is declared without an initializer

# Karpuz

## Comments

Karpuz comments start with %:

```
% this is a comment
```

```
i = 0; % this is also a comment
```

- ▶ → A comment starts with % and runs until the end of the line ←

Here is a variable declaration that works:

```
id = 0;
```

Here is another variable declaration that works:

```
id = null;
```

# Karpuz

## Functions

Here is a function definition:

```
fn print_hello(name) {  
    io.print("Hello", name);  
}
```

# Karpuz

## Scope operators

Karpuz scope operators are: `top` and `local`.

```
fn f() {  
    i = 0;      % brings a new local i into scope  
    top j = 1; % brings module-level j into scope  
               % and assigns 1  
    j = i + k; % from here on, all plain names  
}
```

Laws of scope operators:

- ▶  $\rightarrow$  The first assignment to a name defaults to `local`  $\leftarrow$
- ▶  $\rightarrow$  Later assignments can't use an operator  $\leftarrow$
- ▶  $\rightarrow$  `top i = 0;` brings global var. `i` to local scope  $\leftarrow$
- ▶ How to bring a variable into local scope without writing to it?

# Karpuz

## While loop

Here is a cute little while loop:

```
i = 10;  
while (i > 0) {  
    i = i - 1;  
}
```

- ▶ Only boolean values inside the `while` control
- ▶ It's a runtime trap <sup>1</sup> otherwise

---

<sup>1</sup>*cf.* WASM traps

# Karpuz

## While loop

Here is a not-so-cute little while loop:

```
i = 10;
while (i) {
    i = i - 1;
}
```

- ▶ → This is a runtime trap ←
- ▶ While only supports values of type 'Bool' in its control section

# Karpuz

## labeled break/continue

Karpuz accepts labeled break and labeled continue<sup>2</sup>.

```
while (true) : outer {  
    continue outer;  
    break outer;  
};
```

- ▶ → Any use of break without label should be rejected. ←
- ▶ → Any use of continue without label should be rejected. ←

---

<sup>2</sup>eg. C++ proposal P3568R0

# Karpuz

if

Here is an if statement:

```
i = 10;  
if (i <= 10) {  
    i = i - 1;  
}  
else if (i >= 5){  
    i = i + 1;  
}  
else {  
    i = i * 1;  
}
```

# Karpuz

## Class

Behold this class definition:

```
class C {  
    members = i;  
    fn get_i() {  
        return my.i;  
    }  
}
```

- ▶ All attributes are private
- ▶ All methods are public
- ▶ → Attributes are declared only via `members`. A bare assignment in class scope is a syntax error ←

# Karpuz

## Class

Behold this class definition:

```
class C {  
  members = i;  
  constructor = new;  
  fn new(i) {  
    my.i = i;  
  }  
}  
c := C(4);
```

- ▶ `:=` is the *constructing assignment*: it allocates an instance and calls the method named by `constructor` on it

Laws of the constructing assignment:

- ▶ → The constructor's return value is ignored ←
- ▶ → Without a constructor line, all members start as `null` ←
- ▶ → constructor must name a **method** of the class. Members are not callable ←
- ▶ → The right-hand side of `:=` is a class name, so `x := 5;` is a syntax error ←

# Karpuz

## Module

Behold this module:

```
import io;

class Application {
  members = i;
  fn main(args) {
    return my.i;
  }
}
```

- Zero or more import statements, then fn and class statements

Assignment is an expression. Its value is the assigned value.

```
a = 1; % initialization, returns 1
```

```
a = 2; % returns 2
```

```
b = a = 3; % both a and b are 3
```

# Recap

## Modules

Modules are made of:

- ▶ Zero or more `import` statements.
- ▶ Zero or more `fn` statements.
- ▶ Zero or more `class` statements.

# Recap

## Classes

Classes are made of:<sup>3</sup>

- ▶ Zero or more literals defined as members - optional
- ▶ Constructor statement - optional
- ▶ Method statement - optional

```
class C {  
  members = a, b;  
  constructor = lalala;  
  fn lalala() {...}  
}
```

---

<sup>3</sup>this needs to be further elaborated, eg. see the next slide

# Recap

## Functions

Functions are made of:

- ▶ One KW\_FUNC
- ▶ One IDENTIFIER
- ▶ One OP\_LPAREN
- ▶ Zero or more *identifiers* delimited by OP\_COMMA
- ▶ One OP\_RPAREN
- ▶ One function scope

# Recap

## Scopes

Two types:

- ▶ Function scope
- ▶ Class scope

# Recap

## Function scope

Function scopes are made of the following statement types<sup>4</sup>

- ▶ variable declerations
- ▶ if statement
- ▶ while statement
- ▶ return statement
- ▶ ...
- ▶ Regular statement

---

<sup>4</sup>This list is incomplete