Compiler Design

INF-400

Burak Arslan
ext-inf400@burakarslan.com

Galatasaray Üniversitesi

Lecture III, Part II
2026-10-02

Course website

burakarslan.com/inf400

Karpuz

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

Karpuz

Language Definition

Karpuz

Laws of Karpuz

Note the following notation:

- $\rightarrow$ This is a law of Karpuz $\leftarrow$

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

Karpuz

Variables

Here is a variable declaration that *could* work.

```
id;
```

Karpuz

Variables

Here is a variable declaration that *could* work.

```
id;
```

But it's pointless so we don't allow it. A variable comes into existence only by assignment, so `id;` reads a name that is not bound.

- $\rightarrow$ This is a compile-time error $\leftarrow$
- `Identifier 'id' is not found`

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 ←

Karpuz

Variables

Here is a variable declaration that works:

```
id = 0;
```

Karpuz

Variables

Here is another variable declaration that works:

```
id = null;
```

Karpuz

Functions

Here is a function definition:

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

Karpuz

Scope operators

Karpuz scope operators are: `far` and `near`.

```
j = 0;
fn f() {
  i = 0;      % brings a new local i into scope
  far j;      % j is the module-level j from here on
  j = i + 1; % assigns to the module-level j
};
```

Karpuz

Scope operators

Laws of scope operators:

- → The first assignment to an unbound name creates a local ←
- → `far i;` binds `i` to the module-level `i` ←
- → `near i;` binds `i` to the local `i` of the nearest enclosing function that has one ←
- → A scope operator comes before any local assignment to the same name ←
- Reading a name needs no operator: it resolves to the nearest visible binding

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 [1] otherwise

[1] *cf.* WASM traps

Karpuz

While loop

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

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

- $\rightarrow$ This is a runtime trap $\leftarrow$
- The value of `i` is an `Integer64`, not a `Boolean`

Karpuz

labeled break/continue

Karpuz accepts `labeled break` and `labeled continue`[2].

```
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. ←

[2] *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 ←
- → Inside a method, members and methods are reached only

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

Karpuz

Class

Laws of the constructing assignment:

- → The constructor's return value is ignored ←
- → All members start as `null`, then the constructor runs ←
- → Without a `constructor` line, no method is called ←
- → `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;
  };
};
```

- `import`, `class`, `fn`, assignment and instantiation statements, in any order

Karpuz

Operators

Assignment is a statement, not a value. Assignments can be chained.

```
a = 1;        % initialization
a = 2;        % assigns 2 to a
b = a = 3;    % both a and b are 3
c = (a = 4); % syntax error
```

Recap

Modules

Modules are made of the following statements, in any order:

- `import` statements.
- `fn` statements.
- `class` statements.
- Assignments and instantiations.

Recap

Classes

Classes are made of:[3]

- Identifiers declared by `members` - optional
- Constructor statement - optional
- Method statements - optional

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

[3] this needs to be further elaborated, eg. see the next slide

Recap

Functions

Functions are made of:

- One `KW_FN`
- One `IDENTIFIER`
- One `OP_LPAREN`
- Zero or more *identifiers* delimited by `OP_COMMA`
- One `OP_RPAREN`
- One function scope

Recap

Scopes

Scope types:

- Function scope
- Class scope
- Method scope
- Module scope

Recap

Function scope

Function scopes are made of the following statement types[4]

- Assignments and instantiations
- `if` statement
- `while` statement
- `return` statement
- `...`
- Regular statement

[4] This list is incomplete