Compilers
INF-400

Burak Arslan
ext-inf400@burakarslan.com

Galatasaray Üniversitesi

Lecture VI
2023-11-09

Course website

burakarslan.com/inf400

Kiraz

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

Kiraz

Language Definition

Kiraz
Laws of Kiraz

Note the following notation:
- $\rightarrow$ This is a law of Kiraz $\leftarrow$

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

Kiraz

Variables

Here is a variable declaration that *could* work.

```
let id;
```

Kiraz

Variables

Here is a variable declaration that *could* work.

```
let id;
```

But it'd be too costly.

- $\rightarrow$ This is a compile-time error $\leftarrow$
- Unable to determine the type of 'id'

Kiraz

Variables

Here is a variable declaration that works:

```
let id = 0;
```

Kiraz

Variables

Here is a variable declaration that works:

```
let id = 0;
```

... only when the type of integer literals is well-defined.

- ▶ → The type of id int64 ←
- ▶ Because the type of L_INTEGER(b, 0) is int64

Kiraz

Variables

Here is another variable declaration that works:

```
let id : Int64;
```

Kiraz

Variables

Here is another variable declaration that works:

```
let id : Int64;
```

... but we need a well-defined behavior for uninitialized types.

- ▶ → The value of id is `void` ←
- ▶ Because an *uninitialized* variable is different from a *default-initialized* variable.

Kiraz

Variables

Here is another variable declaration:

```
let id : Int64 = void;
```

- ▶ → This is a compile-time error ←
- ▶ Because a variable can't be initialized to an uninitialized state.

Kiraz

Variables

Here is another variable declaration:

```
let id : Int64 = void;
```

- ▶ → This is a compile-time error ←
- ▶ Because a variable can't be initialized to an uninitialized state.
- ▶ Is it a good idea to disallow this?

Kiraz
Variables

Here is another variable declaration:

```
let id : Int64 = void;
```

- ▶ → This is a compile-time error ←
- ▶ Because a variable can't be initialized to an uninitialized state.
- ▶ Is it a good idea to disallow this?
- ▶ Maybe let's also disallow types that start with lowercase letters?

Structured Programming

Let's go back to sixties ...

Structured Programming

Behold the following C fragments:

```
int main() {
  for (int i = 0; i < 10; ++i) {
    printf("Hello %d\n", i);
  }
  return 0;
}
```

```
int main() {
  int i = 0;
loop_begin:
  printf("Hello %d\n", i);
  if (i >= 10) {
    goto loop_end;
  }
  ++i;
  goto loop_begin;
loop_end:
  return 0;
}
```

Structured Programming

```c
void print_hello(int c) {
  for (int i = 0; i < c; ++i) {
    printf("Hello %d\n", i);
  }
  return 0;
}

int main() {
  print_hello(10);
  return 0;
}
```

Structured Programming

```
int main() {
  static int arg1,arg2,arg3;

  arg1 = 10;
  goto func_print_hello;
.
.
.
func_end_print_hello:
  return 0;
}
```

```
func_print_hello:
  printf("Hello %d\n", i);
  if (i >= 10) {
    goto loop_end;
  }
  ++i;
  goto loop_begin;

loop_end:
  goto func_end_print_hello
```

Structured Programming

- ▶ `break` : goto loop_end;
- ▶ `continue` : goto loop_begin;
- ▶ `return` : goto func_exit;
- ▶ `switch()` : goto given_value;

Structured Programming

Our job is either;

- ▶ Cut costs
- ▶ Increase revenue

Structured Programming

Our job is either;

- ▶ Cut costs
- ▶ Increase revenue
- ▶ Both ☺

Structured Programming

A language that prevents its user from making mistakes is:

- ▶ Cutting costs by lowering development time
- ▶ Increase revenue by accelarating feature delivery

Structured Programming

So it's not like the first time a programming language
prevents actions that the underlying machines
are capable of.

Kiraz
Variables

Here is another variable declaration:

```
let id : Int64 = void;
```

- ▶ → This is a compile-time error ←
- ▶ Because a variable can't be initialized to an uninitialized state.
- ▶ We will think about disallowing types that start with lowercase letters.

Kiraz
Functions

Here is a function definition:

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

- ▶ → This is a parse-time error ←
- ▶ Function 'print_hello' argument 'name' has no type

Kiraz
Functions

Here is a function definition:

```
func print_hello(name : String) {
  io.print("Hello", name);
}
```

- $\rightarrow$ This is a parse-time error $\leftarrow$
- Function 'print_hello' is missing a return type

Kiraz

Functions

Here is a function definition that finally works:

```
func print_hello(name : String) : Void {
  io.print("Hello", name);
}
```

▶ This is a mighty cute little function!

Kiraz

Functions

Here is another function definition:

```
func print_hello(name : String) : Void {
  io.print("Hello", name);
  return 1;
}
```

- ▶ → This is a compile-time error ←
- ▶ Function return value can not be converted to type 'Void'

Kiraz

Functions

Here is another function definition:

```
func print_hello(name : String) : Void {
  io.print("Hello", name);
  return 1;
}
```

- ▶ → This is a compile-time error ←
- ▶ Function return value can not be converted to type 'Void'

Kiraz

Functions

Here is another function definition:

```
func print_hello(name : String) : Int64 {
  io.print("Hello", name);
}
```

▶ Function returns an integer of value void;

Kiraz
Functions

Here is another function definition:

```
func print_hello(name : String) : Int64 {
  io.print("Hello", name);
}
```

▶ Function returns an integer of value void;
▶ What would it take to elevate this to a compile-time error?

Kiraz
While loop

Here is a cute little while loop:

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

▶ Only boolean values inside the `while` control

Kiraz

While loop

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

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

- $\rightarrow$ This is a compile-time error $\leftarrow$
- While only supports values of type 'Bool' in its control section

Kiraz

if

Here is an if statement:

```
let i = 10;
if (i == 10) {
  i = i - 1;
}
else {
  i = i + 1;
}
```

- ▶ Currently, there is no else if statement.
- ▶ Maybe we could add elif (like python)? We will see

Kiraz

Class

Behold this class definition:

```
class C {
  let i : Int64;
  func get_i(): Int64 {
    return i;
  }
}
```

- ▶ All attributes are private
- ▶ All methods are public

Kiraz

Class

Behold this class definition:

```
class C {
  let i : Int64 = 4;
  func get_i(): Int64 {
    return i;
  }
}
```

▶ Attributes can have initializers

Kiraz

Behold this class definition:

```
class C {
  let i : Int64 = 4;
  func new(i: int64): C {
    let retval : C;
    retval.i = i;
    return retval;
  }
}
```

▶ Methods can return own classes;

Kiraz

Module

Behold this module:

```
import io;

class Application {
  let i = 0;
  func main(args: StringArray): Int64 {
    return i;
  }
}
```

- ▶ All attributes are private
- ▶ All methods are public

Recap

Modules

Modules are made of:

▶ Zero or more import statements.

▶ Zero or more `class` statements.

Recap

Classes

Classes are made of:[1]

▶ Zero or more `let` statemens.

▶ Zero or more functions.

Recap

Functions

Functions are made of:

- One `KW_FUNC`
- One `IDENTIFIER`
- One `L_PAREN`
- Zero or more *typed identifiers* delimited by `OP_COMMA`
- One `R_PAREN`
- One `OP_COLON`
- One `IDENTIFIER`
- One function scope

Recap

Scopes

Two types:
- ► Function scope
- ► Class scope

Recap
Function scope

Function scopes are made of the following statement types[2]

- ▶ `let` statement
- ▶ `if` statement
- ▶ `while` statement
- ▶ ...
- ▶ Regular statement

---

[2]This list is incomplete