Ligi:

A Readable Programming Language

Johnathan Lee

Computer Science Information Systems, Minnesota State University Moorhead, 1104 7th Avenue South, Moorhead, MN 56563

Language:

Ligi (Esperanto: “to bind”) is a statically typed language designed to fix perceived problems with other languages, such as C, C++, Java, and C#, and produce more results with less typing and in a logical, readable format.

This poster only scratches the surface of Ligi’s syntax. For more, visit Ligi at github.com/JohnathanFL/Ligi

Variable assignments are done through let, var, and cvar, semicolons are optional, and ‘(:’ denotes comments:

```
let x: usize = 0 (: Constant uint{32 or 64}
var y+: usize = 0 (: Mutable, exported(+) to other scopes
cvar z: usize = (: Compiletime mutable, runtime constant
```

A simple adder function could be written as such:

```
let adder = pure fn a: usize, b: usize → c = a + b
```

This can read in English as “let adder be a pure (predictable) function of a and b, which are usizes, that returns c, which is calculated with ‘a + b’”.

Types are declared using mathematical expressions which act on other types. For example, simple 2- and 3-dimensional points with floating-point precision could be declared as such:

```
let Point2 = struct { field x+: f32, y+: f32 }
let Point3 = Point2 + struct { field z+: f32 }
let point = [:Point3: .x = 10, .y = 0, .z = 20]
```

`struct` is a full unary opertator that evaluates a block as a new struct type, and “field” is a bind-type on the same level as “let” or “var” that declares a new memory location inside a struct. The `[:type:...]` syntax initializes a struct or array.

Adding 2 structs together works similarly to, but is not the same as, inheritance in other languages. This type addition can be expanded to work with enums (which work like tagged unions) to utilize readable error types, rather than ugly exceptions:

```
let BaseResult = enum { enum Ok: usize }
let Result = BaseResult + enum (OutOfMemory, OutOfRange)
let function = fn → res:BaseResult = #Ok(10)
assert function() == #Ok(10) and function() ≠ #OutOfMemory
```

Thus a function which may fail by a memory error can return Result, which is either #Ok, which then contains the function’s real return, or one of the enums #OutOfMemory or #OutOfRange to tell the programmer what went wrong.

Rather than `Type<T>` template syntax, Ligi takes inspiration from Zig[1], where generic types are achieved through functions that return types:

```
let ArrayOf3 = pure fn T: type → result = struct { field data: array (3, T) }
let arOf10Bools = [:ArrayOf10(bool): .data = [true, false, true]]
```

One of Ligi’s flagship features is “Swizzling”, a feature from graphics programming languages such as GLSL[2] that allows selecting multiple members from a struct and acting on them all at once. For example, you could unpack the x, y, and z attributes of an instance of the above Point3 like so:

```
let (x, y, z) = point.(x, y, z)
```

The `.(member1, member2, member3)` syntax returns a tuple of members and the results of function calls found inside the parentheses. `let` statements also natively support unpacking tuples, leading to simplified statements like the above.

Swizzling can also be used to effect method-chaining without having to return a back-reference. For example, to call func1 and func2 on a variable `a` without repeating `a.` over and over:

```
a.(func1(), func2())
```

This is especially useful if, instead of *a,* you had a long expression where you wanted to call multiple methods on the result.

If func1 and func2 returned things, you could then collect them at the same time:

```
let (res1, res2) = a.(func1(), func2())
```

It’s important to note that `object.(func1(), func2()…)` syntax always works on the same object. The functions may mutate *object,* but it is still the same object. If *func1* returns an object, *func2* is not acting on it. For example, if you wanted to chain off of the results of multiple functions, it would instead be written like:

```
object.obj_func().result1_func().result2_func()
```

Format:

When an editor wishes to provide support for a language, it must often write new code from scratch to handle gathering data about the source files from scratch. In recent years, however, the concept of a language server has appeared. Language servers handle the heavy lifting of parsing and storing information about a project, and editors simply make requests to them over the network to get the needed information. Ligujo draws inspiration from, but does not implement, the Language Server[3] protocol.

Ligujo (Esperanto: “that which contains binds”) is a server intended to work in concert with Ligi’s compiler and the programmer by holding and providing information about the types in a Ligi program. Thus, Ligujo fulfills half of a language server’s duty, and the compiler fulfills the other half.

Ligujo lives at github.com/JohnathanFL/Ligujo

Ligujo accepts HTTP-compliant GET and PUT requests.

The compiler could use PUT requests to add new types to the server. For example, a compiler could request *PUT /mktype* with the following JSON structure as a body to register the previous “Point2” type with Ligujo:

```
{
  - The ID of the type, as determined by the compiler
  “431136”: {
    “name”: “Point2”, - The name of the type
    “pos”: “6:5”, - The line and column it was declared at
    ”ty”: 2058, - The ID for a `struct`
    “fields”: [ - All of the fields of the struct.
       - Access level 2 is public (private, readonly, public)
       - TypeID 2053 is reserved as a 32-bit float
      { “access”: 2, “name”: “x”, “type”: 2053 },
      { “access”: 2, “name”: “y”, “type”: 2053 }
    ]
  }
}
```

Although no editor is currently equipped to handle Ligujo’s protocol, the intended usage is that an editor uses the GET methods to retrieve information from Ligujo.

For example, an editor in which a user is viewing the following piece of code:

```
1 let x = 0
2 let point = [:Point2: .x = 10.0, .y = 30.0]
```

And we already knew (through some other function) that the typeid of *point* was *431136,* we could send a request to *GET /typeid/431136* from Ligujo to retrieve the type of *point*. Ligujo would then fire back a response (screenshot includes Ligujo’s output for reference):

```
└[$] (master) curl -X GET localhost:8080/typeid/431136
{"name":"Point2","statics":[],"pos":"6:5","ty":2058,"backing":0,"fields":[{"access":2,"name":"
x","typeid":2053},{"access":2,"name":"y","typeid":2053}]}↵

Saluton, mondo!
Handling request...
Retrieving #431136
  Found type Point2 @ 6:5 of ty 2058 backed by 0
    Err was <nil>
    Found field "x" with access 2 containing 2053
    Err was <nil>
    Found field "y" with access 2 containing 2053
```

Expanded out with comments, the response is:

```
{
  “name”: “Point2”,
  “statics”: [], – Point2 has no statics
  “pos”: “6:5”, – Point2 was declared at line 6, column 5
  “ty”: 2058, – Point2 is a `struct`
  “backing”: 0, – Irrelevant for a struct
  “fields”: [ – Point2 has 2 fields: x and y
    {“access”: 2, “name”: “x”, “typeid”: 2053},
    {“access”: 2, “name”: “y”, “typeid”: 2053}
  ]
}
```

Citations:

[1]A. Kelly, "The Zig Programming Language", Ziglang.org, 2020. [Online]. Available: https://ziglang.org/. [Accessed: 17- Apr- 2020].

[2] "Data Type (GLSL) - OpenGL Wiki", Khronos.org, 2020. [Online]. Available: https://www.khronos.org/opengl/wiki/Data_Type_(GLSL)#Swizzling. [Accessed: 17- Apr- 2020].

[3] "Langserver.org", Langserver.org, 2020. [Online]. Available: https://langserver.org. [Accessed: 17- Apr- 2020].