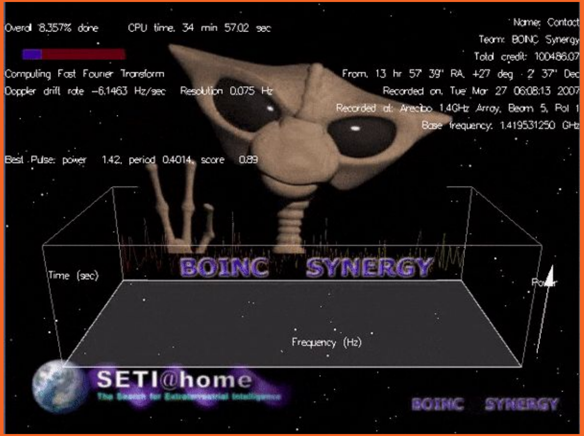
The Outside World



Overall 8.357% done CPU time, 34 min 57.02 sec
Computing Fast Fourier Transform
Doppler drift rate -6.1463 Hz/sec Resolution 0.075 Hz
Best Pulse: power 1.42, period 0.4014, score 0.89

Name: Contact
Team: BOINC Synergy
Total credit: 100486.07
From: 13 hr 57' 39" RA, +27 deg 2' 37" Dec
Recorded on: Tue Mar 27 06:08:13 2007
Recorded at: Arecibo 1.4GHz Array, Beam 5, Pol 1
Base frequency: 1.419531250 GHz

Scryer Prolog Meetup 2023 - Düsseldorf
Adrián Arroyo Calle - https://adrianistan.eu

?- person(Name, Surname).

Name = "Adrián Arroyo Calle".

City = "Valladolid, Spain"

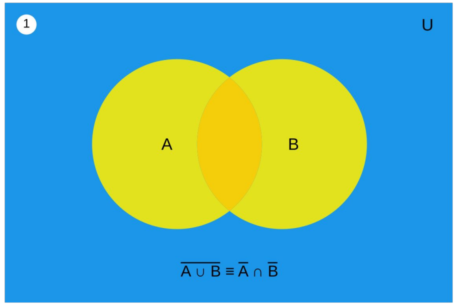
Job = "Backend developer at Telefónica"

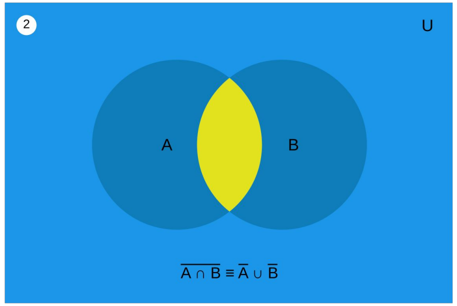
Website = "https://adrianistan.eu"

Mail = "aarroyoc@adrianistan.eu"

Mastodon = "aarroyoc@castilla.social"

My Prolog background





Always liked classic logic. I learnt about it first in high school in philosophy class (truth tables, Modus Tollens, De Morgan laws, …)

My first contact in Prolog was in university. At first I didn’t understand it. But it was a challenge!

I end up liking the challenge. I like how you could focus more on the description of the problem. I started to use Prolog more and more for “general” programming. But Prolog can improve a lot!

There are lot of things in Prolog that can be improved. I'll focus on "the Outside World"

Turing complete is not enough

Prolog is a Turing complete language, that means every problem that can be computed in any mainstream language (think Java, Rust, Python, C, …) can be expressed in Prolog too.

However, when we try to do some stuff in Prolog is… *complicated*

Why it's not enough?

Operating systems do not talk Prolog

Hardware does not talk Prolog

Lots of mature software do not talk Prolog (we don't want to throw up the massive work done by millions of people!)

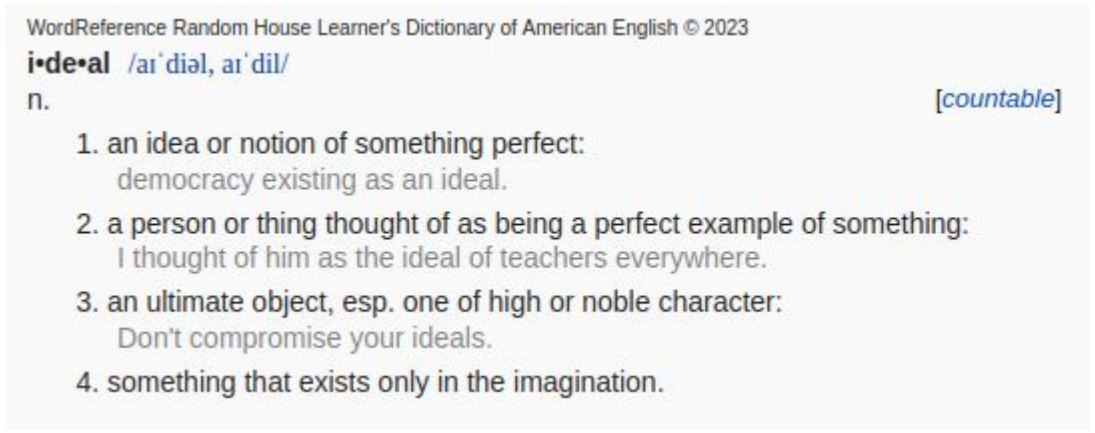
For lots of projects Prolog is not a good tool *yet*

Even today, most people is not coding in Prolog! So every day, *excellent* code gets written in \+ Prolog and we can't use it :/

The ideal Prolog system

The only part that really talks Prolog is the Prolog system!

An ideal Prolog system should hide all the details so that we only talk Prolog yet we can do whatever we want! *Ideally it would be an OS by itself*


WordReference Random House Learner's Dictionary of American English © 2023

i•de•al /aɪˈdiəl, aɪˈdil/

n. [*countable*]

1. an idea or notion of something perfect:
   democracy existing as an ideal.
2. a person or thing thought of as being a perfect example of something:
   I thought of him as the ideal of teachers everywhere.
3. an ultimate object, esp. one of high or noble character:
   Don't compromise your ideals.
4. something that exists only in the imagination.

Inside Out ; Out Inside

So, we need to talk to the outside world. The Prolog system should provide mechanisms to do that talk. There are two types of mechanisms:

Outside -> Prolog	**Prolog -> Outside**
Library embedding (WASM too)	Files and sockets
Files and sockets	Foreign Function Interface
Execute scryer-prolog in subshell	

Files and sockets

Files can be used to communicate with the operating system and other programs. Some systems such as Linux actively promote this usage.

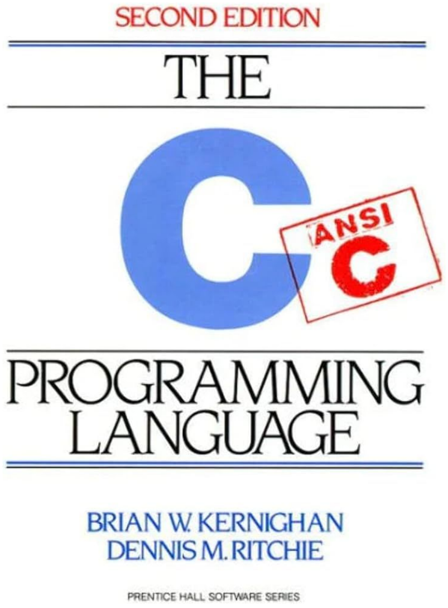
Sockets are files too! And they can go from one machine to another. If we do not want to manage low level details we can use HTTP, which started for the WWW but now it's used for APIs in all kind of systems. Some systems also provide IPC like D-Bus on desktop Linux but it is not very portable.

Why not files then?

Files and sockets provide good solutions for interacting with the outside world, and they’re already supported by Scryer Prolog. However they have a few problems:

- There must be something reading/listening at the other side!
- Each application should define its own DSL to interact with
- Poor error handling
- Performance and latency

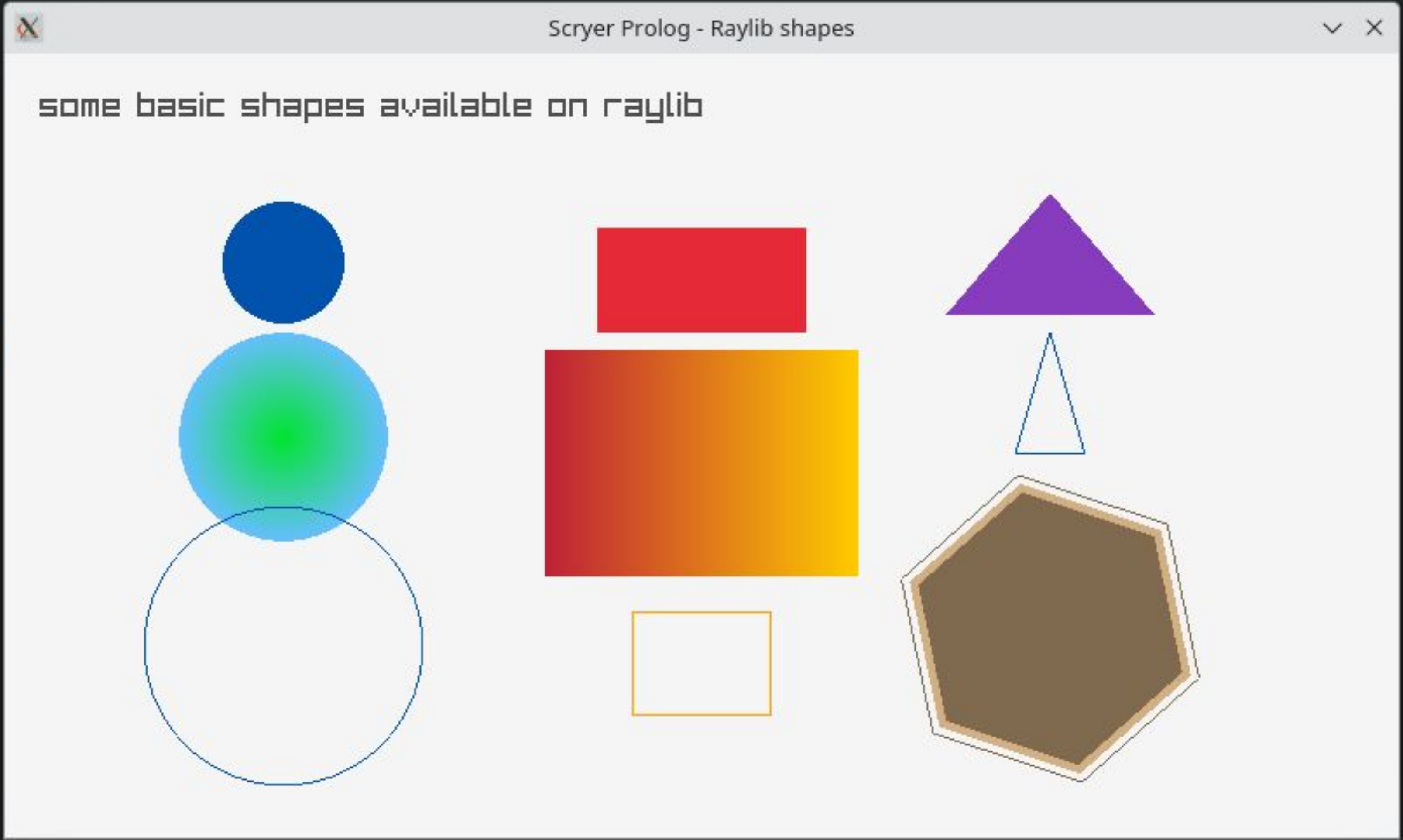
FFI, or, being an actor

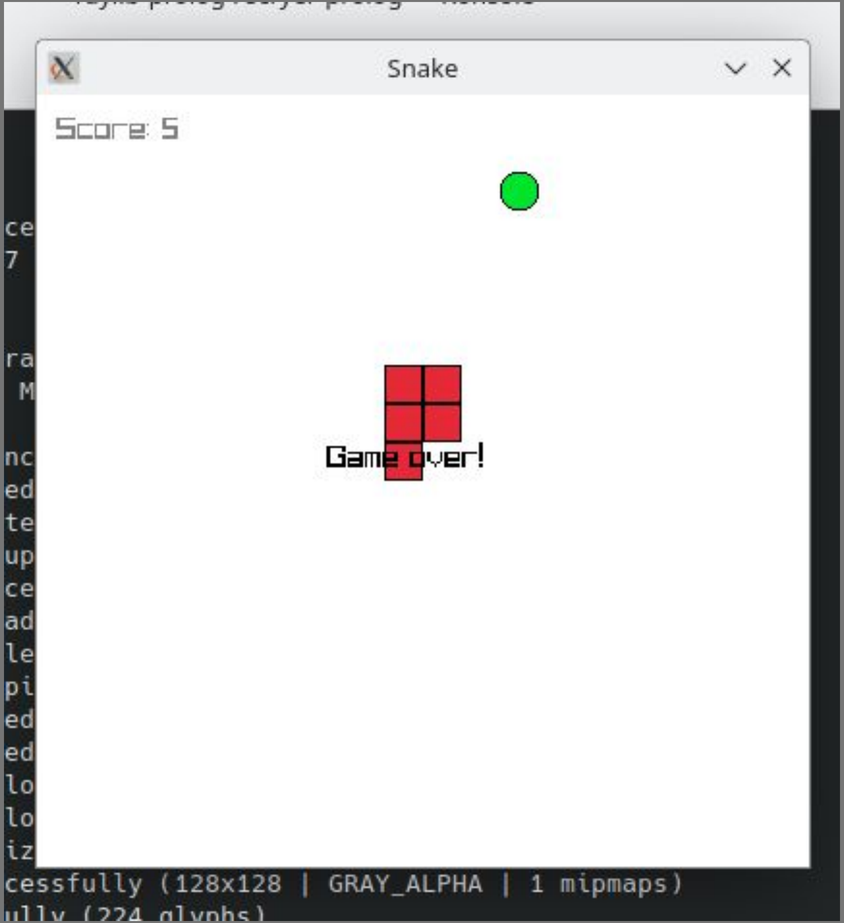


Foreign Function Interface could be resumed as *if language X is doing that to call function Y, we replicate that behaviour and we could call Y just as if we were programming in X!*

X always bounds to C for all practical purposes.

What has the FFI ever done for us?


Scryer Prolog - Raylib shapes
some basic shapes available on raylib



```
Score: 5

Game over!
```

```
ce
7

ra
 M

nc
ed
te
up
ce
ad
le
pi
ed
ed
lo
lo
iz
cessfully (128x128 | GRAY_ALPHA | 1 mipmaps)
ully (224 glyphs)
```

How does it work?

Let’s do an example from scratch!

First, the principles

The current design of FFI has been designed using the following principles:

- No need for glue code
- Should be as declarative as possible
- Should be similar to current module system
- Should be simple

Trealla Prolog did an initial version of this. I just modified a little bit the design.

use_foreign_module

Similar to use_module, to interop with a native library we use use_foreign_module.

- First argument must be the location of the native library (.dll, .so, .dylib)
- Second argument is a list of functions that we want to import. We need the name of the function, the input argument types and the output type.

Types

We define some basic types to be used in the function import definitions: void, bool, f32, f64, sint8, sint16, sint32, sint64, uint8, uint16, uint32, uint64, ptr.

Those are the building blocks. They fit into registers. They can point to memory addresses (useful for pointers).

We additionally allow defining custom structs: foreign_struct. A struct in C is a pack of other types.

There’s an additional cstr type that allows us to use Prolog string directly doing additional conversions.

foreign_struct

To define a custom struct, we declare a name and the type of each property. Names do not matter, but the order matters!

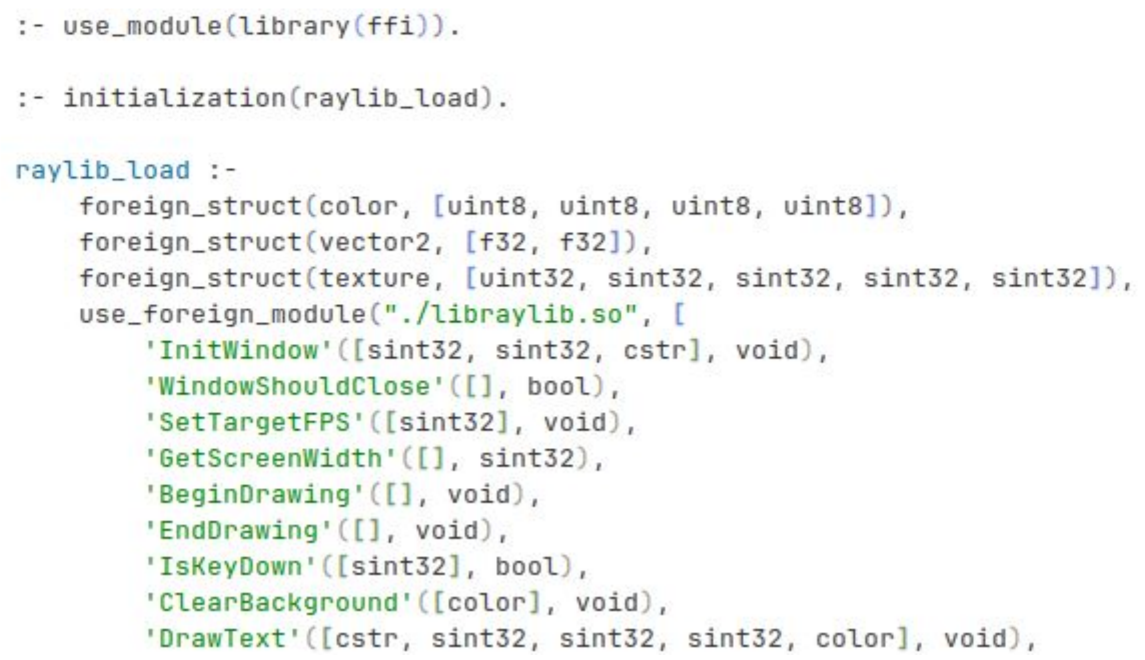
For example:

```
?- foreign_struct(color, [uint8, uint8, uint8]).

struct {

   uint8_t red; uint8_t green; uint8_t blue;

} // not HDR friendly… :)
```

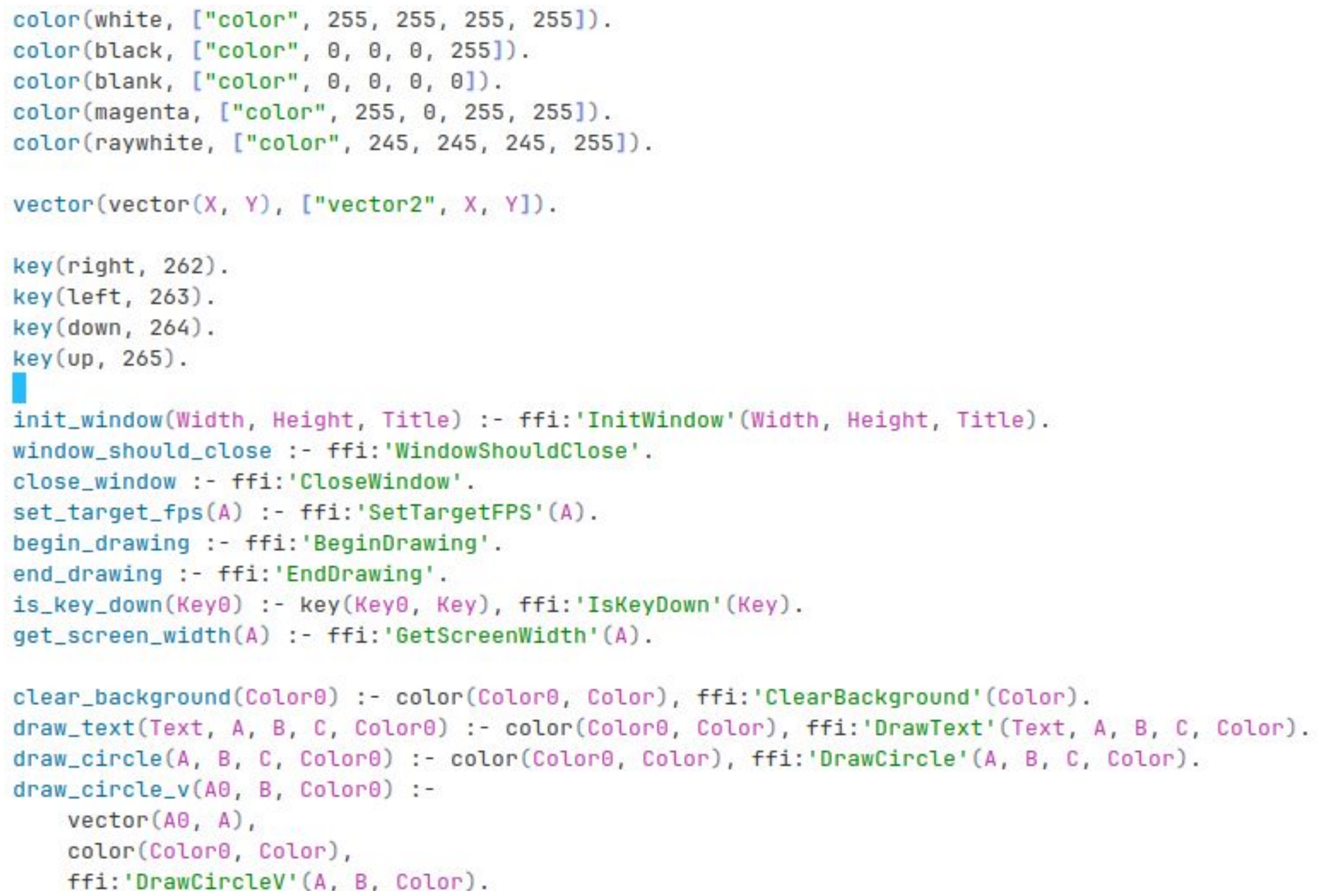
For each function in the import list we assert a predicate with the same name in the module ‘ffi’. Each input argument gets mapped in order. The last argument is the output argument. However, two exceptions:

- functions that return void do not have an output argument
- functions that return bool do not have an output argument but the predicate fails if the function returns false

All together (Raylib)



```
:- use_module(library(ffi)).

:- initialization(raylib_load).

raylib_load :-
    foreign_struct(color, [uint8, uint8, uint8, uint8]),
    foreign_struct(vector2, [f32, f32]),
    foreign_struct(texture, [uint32, sint32, sint32, sint32, sint32]),
    use_foreign_module("./libraylib.so", [
        'InitWindow'([sint32, sint32, cstr], void),
        'WindowShouldClose'([], bool),
        'SetTargetFPS'([sint32], void),
        'GetScreenWidth'([], sint32),
        'BeginDrawing'([], void),
        'EndDrawing'([], void),
        'IsKeyDown'([sint32], bool),
        'ClearBackground'([color], void),
        'DrawText'([cstr, sint32, sint32, sint32, color], void),
```



```
color(white, ["color", 255, 255, 255, 255]).
color(black, ["color", 0, 0, 0, 255]).
color(blank, ["color", 0, 0, 0, 0]).
color(magenta, ["color", 255, 0, 255, 255]).
color(raywhite, ["color", 245, 245, 245, 255]).

vector(vector(X, Y), ["vector2", X, Y]).

key(right, 262).
key(left, 263).
key(down, 264).
key(up, 265).

init_window(Width, Height, Title) :- ffi:'InitWindow'(Width, Height, Title).
window_should_close :- ffi:'WindowShouldClose'.
close_window :- ffi:'CloseWindow'.
set_target_fps(A) :- ffi:'SetTargetFPS'(A).
begin_drawing :- ffi:'BeginDrawing'.
end_drawing :- ffi:'EndDrawing'.
is_key_down(Key0) :- key(Key0, Key), ffi:'IsKeyDown'(Key).
get_screen_width(A) :- ffi:'GetScreenWidth'(A).

clear_background(Color0) :- color(Color0, Color), ffi:'ClearBackground'(Color).
draw_text(Text, A, B, C, Color0) :- color(Color0, Color), ffi:'DrawText'(Text, A, B, C, Color).
draw_circle(A, B, C, Color0) :- color(Color0, Color), ffi:'DrawCircle'(A, B, C, Color).
draw_circle_v(A0, B, Color0) :-
    vector(A0, A),
    color(Color0, Color),
    ffi:'DrawCircleV'(A, B, Color).
```

It should be possible to autogenerate this Prolog glue from C headers. *Do we want it?*

Now, the magic

And why we must use libffi

Let's see the Rust code that implements this

dlopen / LoadLibrary

- When the user defines a foreign_module, the first thing to do is open the library itself. This API is different on each operating system. We use libloading crate to do it cross platform
- For each function defined we search for the entry point of the function (CodePtr). We use the names of the functions, in C it is defined to keep the same name as defined in the code, but not in C++ or Rust! (we must compile with extern “C”)

libffi

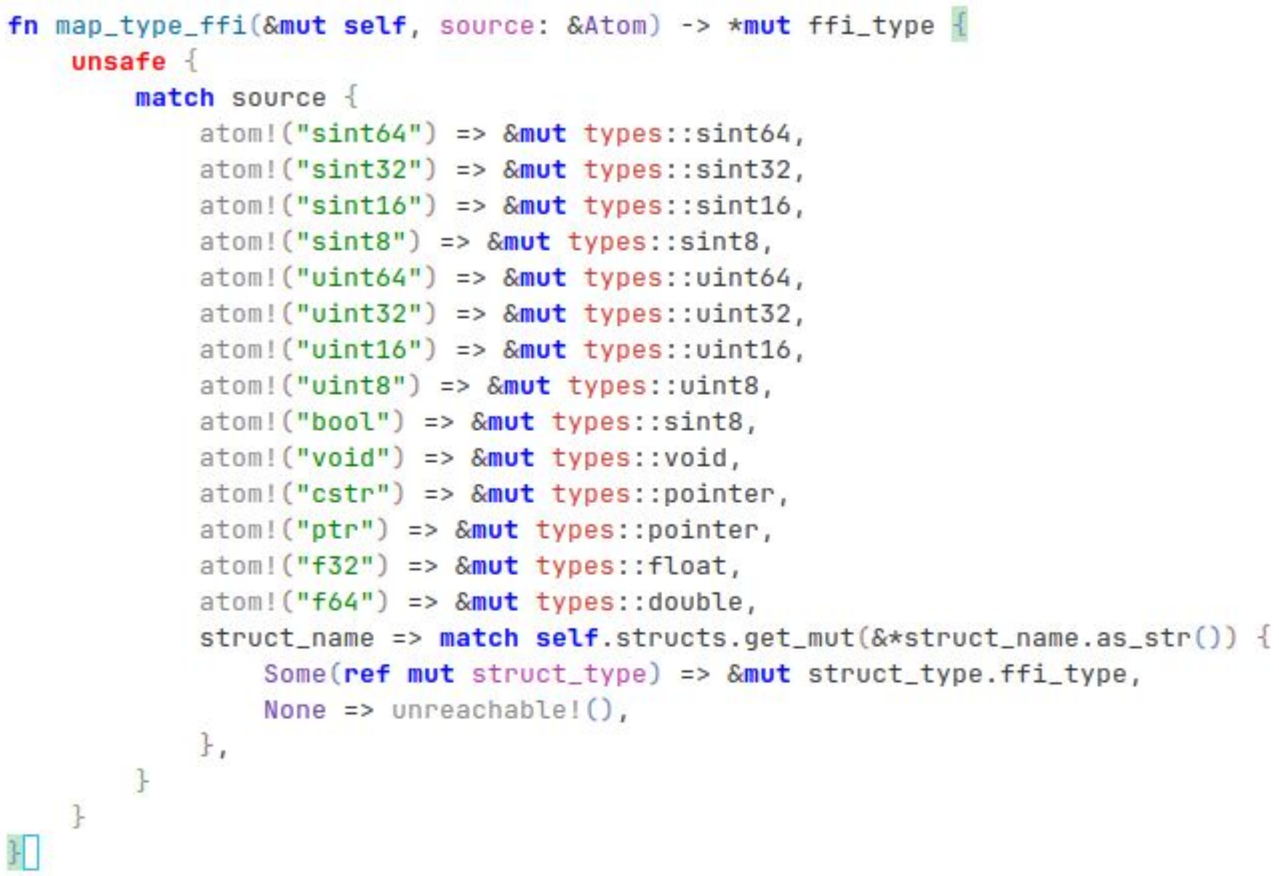
To call a function with FFI we must follow a calling convention. Those conventions are different in different operating systems **and** architectures. libffi is a battle tested library that detects and implements that calling conventions for us.

Used by:

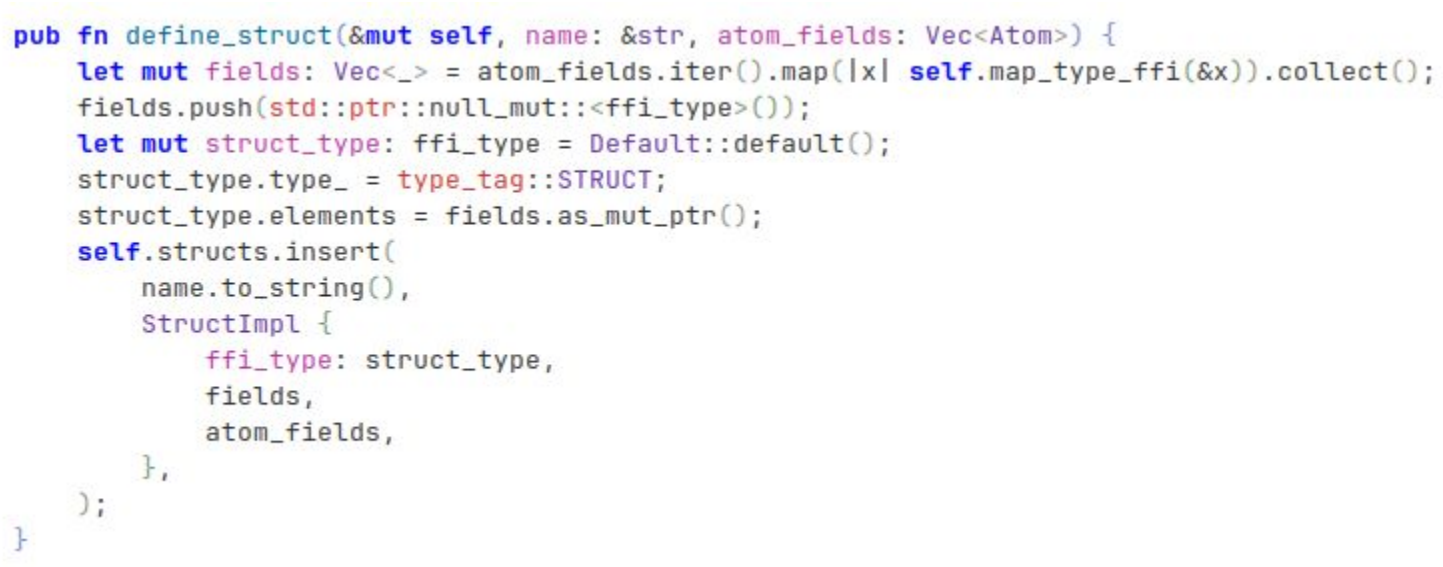
- CPython
- OpenJDK (Java)
- GHC (Haskell)
- Racket
- Deno (JavaScript)
- and more…

The basic building block of libffi is the CIF. A CIF is a definition of a foreign function for an specific ABI. In a CIF we must define input arguments and the output argument. They must be ffi_type.

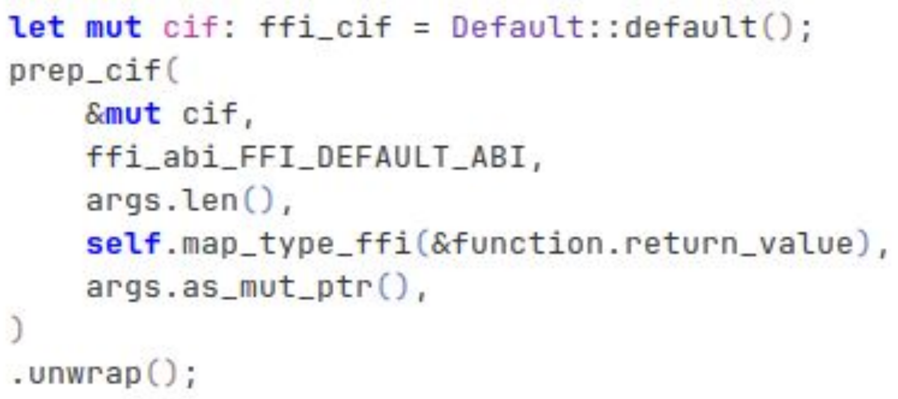
There's a map between Prolog FFI types and libffi ffi_types. Also for structs.



```rust
fn map_type_ffi(&mut self, source: &Atom) -> *mut ffi_type {
    unsafe {
        match source {
            atom!("sint64") => &mut types::sint64,
            atom!("sint32") => &mut types::sint32,
            atom!("sint16") => &mut types::sint16,
            atom!("sint8") => &mut types::sint8,
            atom!("uint64") => &mut types::uint64,
            atom!("uint32") => &mut types::uint32,
            atom!("uint16") => &mut types::uint16,
            atom!("uint8") => &mut types::uint8,
            atom!("bool") => &mut types::sint8,
            atom!("void") => &mut types::void,
            atom!("cstr") => &mut types::pointer,
            atom!("ptr") => &mut types::pointer,
            atom!("f32") => &mut types::float,
            atom!("f64") => &mut types::double,
            struct_name => match self.structs.get_mut(&*struct_name.as_str()) {
                Some(ref mut struct_type) => &mut struct_type.ffi_type,
                None => unreachable!(),
            },
        }
    }
}
```



```rust
pub fn define_struct(&mut self, name: &str, atom_fields: Vec<Atom>) {
    let mut fields: Vec<_> = atom_fields.iter().map(|x| self.map_type_ffi(&x)).collect();
    fields.push(std::ptr::null_mut::<ffi_type>());
    let mut struct_type: ffi_type = Default::default();
    struct_type.type_ = type_tag::STRUCT;
    struct_type.elements = fields.as_mut_ptr();
    self.structs.insert(
        name.to_string(),
        StructImpl {
            ffi_type: struct_type,
            fields,
            atom_fields,
        },
    );
}
```



```
let mut cif: ffi_cif = Default::default();
prep_cif(
    &mut cif,
    ffi_abi_FFI_DEFAULT_ABI,
    args.len(),
    self.map_type_ffi(&function.return_value),
    args.as_mut_ptr(),
)
.unwrap();
```

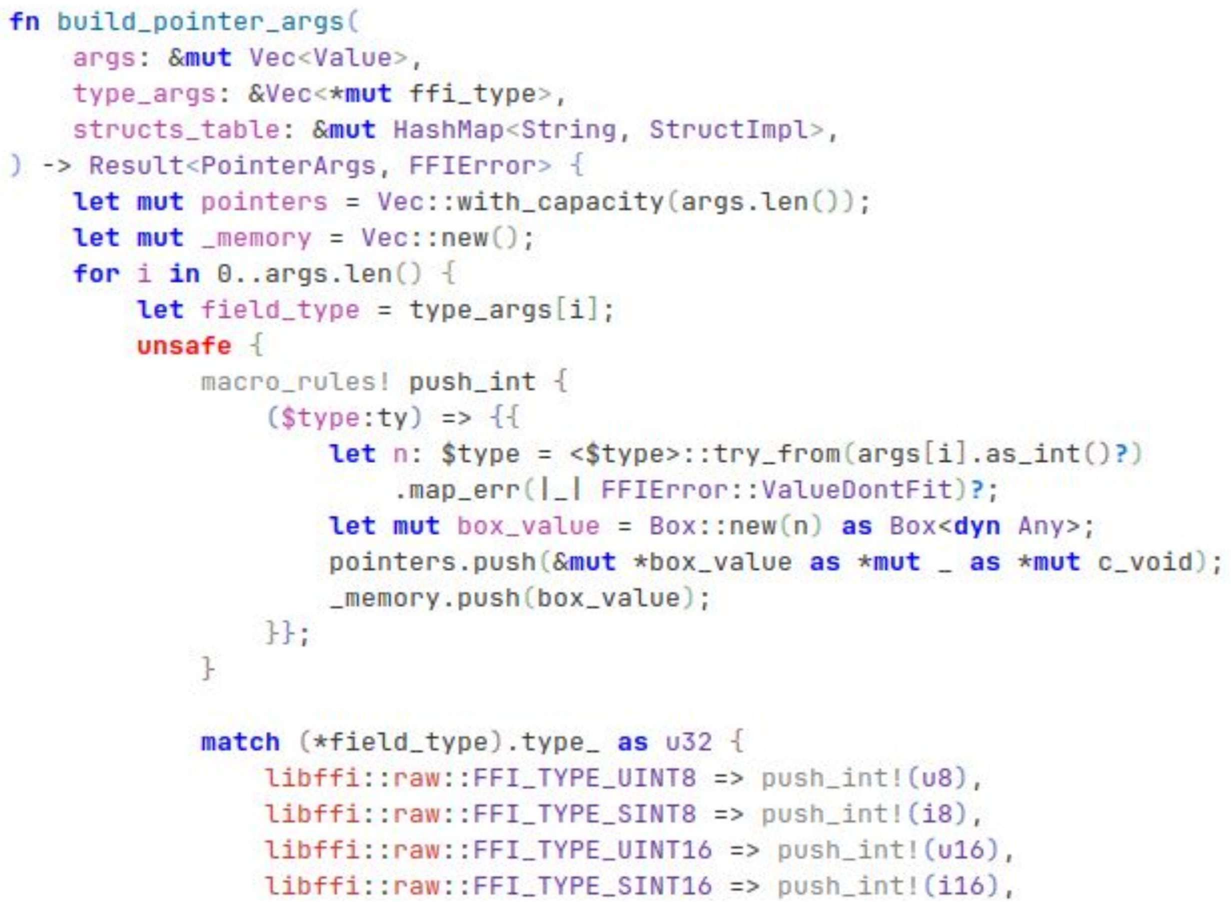
We store the CIF along with the CodePtr and some other data to facilitate conversions between Prolog and the ABI.

Call me, maybe?

We're now ready to explain what happens when we call a native function. We'll see that due to the amount of work we need to do on every call these calls are expensive.

Once we find the CIF and CodePtr for the function we must map each argument to a pointer. libffi does not accept arguments directly, we must pass a pointer to the data. We can use Box to put data in the heap and get a raw pointer (*mut c_void)

We store the Boxes in a Vec so when the Vec drops, we liberate the memory from the arguments too!

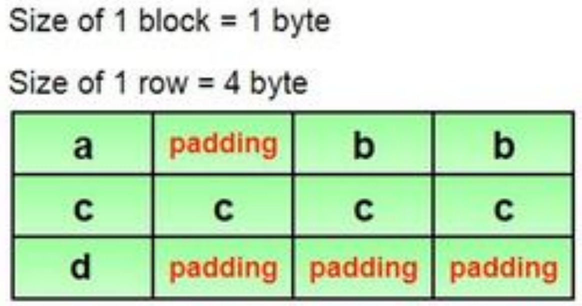


```rust
fn build_pointer_args(
    args: &mut Vec<Value>,
    type_args: &Vec<*mut ffi_type>,
    structs_table: &mut HashMap<String, StructImpl>,
) -> Result<PointerArgs, FFIError> {
    let mut pointers = Vec::with_capacity(args.len());
    let mut _memory = Vec::new();
    for i in 0..args.len() {
        let field_type = type_args[i];
        unsafe {
            macro_rules! push_int {
                ($type:ty) => {{
                    let n: $type = <$type>::try_from(args[i].as_int()?)
                        .map_err(|_| FFIError::ValueDontFit)?;
                    let mut box_value = Box::new(n) as Box<dyn Any>;
                    pointers.push(&mut *box_value as *mut _ as *mut c_void);
                    _memory.push(box_value);
                }};
            }

            match (*field_type).type_ as u32 {
                libffi::raw::FFI_TYPE_UINT8 => push_int!(u8),
                libffi::raw::FFI_TYPE_SINT8 => push_int!(i8),
                libffi::raw::FFI_TYPE_UINT16 => push_int!(u16),
                libffi::raw::FFI_TYPE_SINT16 => push_int!(i16),
```

But with structs...

To pass value structs we must do additional stuff. We need to **align** the struct. It means we need to pack the data in a certain way. This is system dependant.

Luckily, libffi generates that data for structs for us. And Rust has an API to write memory following alignments!

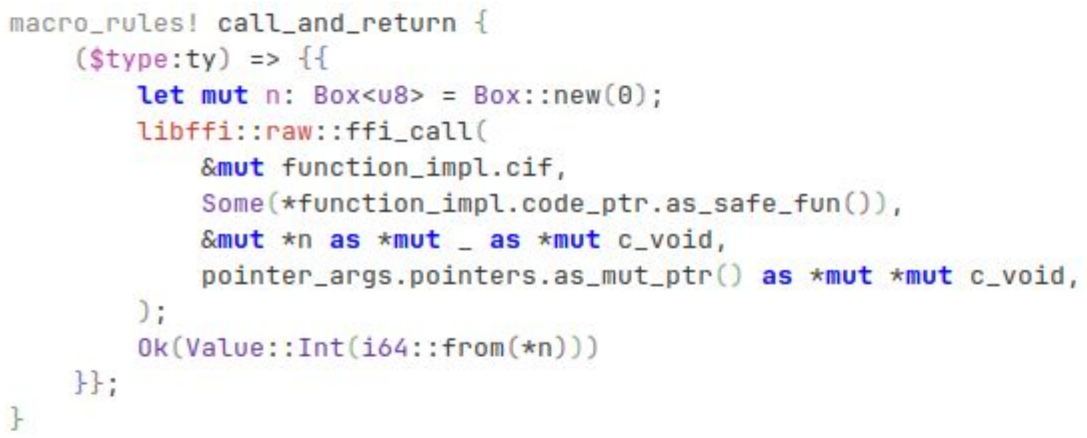


Size of 1 block = 1 byte

Size of 1 row = 4 byte

a	padding	b	b
c	c	c	c
d	padding	padding	padding

Finally, we can ffi_call



```rust
macro_rules! call_and_return {
    ($type:ty) => {{
        let mut n: Box<u8> = Box::new(0);
        libffi::raw::ffi_call(
            &mut function_impl.cif,
            Some(*function_impl.code_ptr.as_safe_fun()),
            &mut *n as *mut _ as *mut c_void,
            pointer_args.pointers.as_mut_ptr() as *mut *mut c_void,
        );
        Ok(Value::Int(i64::from(*n)))
    }};
}
```

But for structs we need to read a block of memory, following the same alignment as when writing parameters.

Ok, but what about arrays?

Fixed size arrays are the same as structs, but all the fields are of the same type. We don’t have an specific way to define them and currently requires to register them with a name. *Should we change this?*

Some libraries are harder than others

Let’s take for example SDL:

- Almost all require you to allocate memory first for structs and their functions only set the data. *(There’s a PR with one possible solution)*
- Sometimes they don't use structs but unions!
- Some functions don't really exist, they're just C macros. The binding process needs to reconstruct what the macro is doing.

More missing stuff

- Callbacks
  - Certain APIs like wgpu require being able to pass a pointer to a function they are going to call. Right now, it seems very difficult to do it.
- More memory manipulation helpers
  - While we can call malloc/free, we cannot manipulate raw data easily: think array[12] = SOMETHING.
  - Modifying fields in structs when we have a pointer
- Improve errors
- An adaptation for WASM?
- Better tooling?
- Is this the right approach? Or a plugin based approach makes more sense?

E END IS NEVER THE END IS NEVER THE END IS NEVER THE END IS **LOADING** NEV