Sand

Sand is a sandbox for Elixir. It is probably better than the other sandboxes for Elixir. **Nonetheless, it should not be used at this point in time.**

Sand employs AST whitelisting and BEAM features to ensure that untrusted Elixir code can be run without side effects, that memory and CPU usage are limited, and that the atom table is not filled.

```
Sand.run("""
r factorial = fn
  1 -> 1
  n -> n * factorial.(n - 1)
end

factorial.(22222)
""")

# returns {:error, :killed}
```

Well-behaved programs run without a hitch:

```
Sand.run("""
r factorial = fn
  1 -> 1
  n -> n * factorial.(n - 1)
end

factorial.(5)
""")

# returns {:ok, 120, %Sand{...}}
```

The Whitelist

Sand operates a small whitelist of permitted language constructs. Everything else is forbidden.

Only a subset of Elixir language constructs is available. Additionally, user code cannot access named functions (of the form `Module.function`) except for a small number of pre-imported functions, such as `+/2` and `is_integer/1`. This means that the Elixir standard library is not available in the sandbox. Functions without side effects, such as `Enum.map`, can, however, be re-implemented as anonymous functions in the sandbox.

These are the permitted language constructs:

- The following reserved keywords: `do and or end in true false nil when not else fn`
- The following inlined operators: `== != > >= < <= + - * / ++ --`
- The inlined `is_*` operators, used for type checking
- These language constructs: `= {} -> %{} | &`

- Variables, atoms, integers, floats, lists, anonymous functions and anonymous function calls
- Binary strings smaller than 64 bytes
- These macros: `case ^ |> if in`
- The recursion macro `r`, shown in the examples above, which enables recursion in anonymous functions

Resource Management

Resource management is done at the level of the Erlang VM, not the OS.

By default, a sandbox process will be killed if it uses more than 1 MB in memory, if it performs more than 1 million reductions, or if it runs for more than 10 seconds. Note that these limits are soft: the process is killed only when the monitor notices the transgression. One should anticipate disobedient processes to briefly use slightly more resources than allowed before being killed.

Memory is limited using `max_heap_size`. Computation is limited by monitoring `Process.info(:reductions)`. Run time is limited using `:timer.exit_after`.

State

Both state and configuration are held in the `%Sand{...}` struct. This struct can be passed as the first argument to `Sand.run/2` and `Sand.assign/2`. Each of those functions returns an altered `Sand` struct, holding the new state.

Like IEx, Sand stores all top-level variables in the state:

```
{:ok, 9, box} = Sand.run("""
squares = %{3 => 9, 4 => 16, 5 => 25}
squares[3]
""")

{:ok, 16, _} = Sand.run(box, "squares[4]")
%{3 => 9, 4 => 16, 5 => 25} = Sand.get(box, "squares")
```

Assigning globals can also be done programmatically. This is useful for providing input to user code:

```
user_code = "input[:width] * input[:height]"

Sand.assign("input", %{width: 50, height: 100})
|> Sand.run(user_code)
```

Configuration

The majority of fields in the `%Sand{...}` struct are configuration options:

```
max_heap_size: 125_000, # process memory in words
max_reductions: 1_000_000, # maximum number of reductions per call of Sandbox.run/2
```

```
max_vars: 10_000, # Maximum number of variables
timeout: 10_000 # Number of milliseconds before Sandbox.run/2 is aborted
```

These can be altered between runs:

```
lil_memory = %Sand{max_heap_size: 10_000}
{:ok, res, new_box} = Sandbox.run(lil_memory, some_user_code)
more_memory = %{ new_box | max_heap_size: 1_000_000}
{:ok, res2, final_box} = Sandbox.run(more_memory, demanding_user_code)
```

The Recursion Macro

Elixir does not support recursion in anonymous functions:

```
loop = fn -> loop.() end

# ** (CompileError) iex:8: undefined function loop/0
#     (elixir 1.10.4) src/elixir_fn.erl:15: anonymous fn/4 in :elixir_fn.expand/3
#     (stdlib 3.13) lists.erl:1354: :lists.mapfoldl/3
#     (elixir 1.10.4) src/elixir_fn.erl:20: :elixir_fn.expand/3
```

Since anonymous functions are the only permitted functions, the Z-combinator macro `r` is inlined:

```
# The syntax is `r [NAME] = fn [BODY] end`

r loop = fn -> loop.() end
```

Installation

The package can be installed by adding `sand` to your list of dependencies in `mix.exs`:

```
def deps do
  [
    {:sand, "~> 0.1.0"}
  ]
end
```