**Types**

<table>
<thead>
<tr>
<th>Type</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer</td>
<td><code>1234</code> <code>0xc0f7</code> <code>0177</code> <code>0b100</code> <code>10.0</code> <code>000</code></td>
</tr>
<tr>
<td>Float</td>
<td><code>1.0</code> <code>3.1415</code> <code>6.02e23</code></td>
</tr>
<tr>
<td>Atom</td>
<td><code>:foo</code> <code>me@home</code> <code>with spaces</code></td>
</tr>
<tr>
<td>Tuple</td>
<td><code>{ 1, 2, :ok, &quot;xy&quot; }</code> (like array)</td>
</tr>
<tr>
<td>List</td>
<td><code>[ 1, 2, 3 ]</code> (like linked list)</td>
</tr>
<tr>
<td></td>
<td>`[ head</td>
</tr>
<tr>
<td></td>
<td><code>'abc'</code></td>
</tr>
<tr>
<td></td>
<td>&quot;&quot; &quot;here doc&quot; &quot;&quot;</td>
</tr>
<tr>
<td></td>
<td>(see Enum and List modules)</td>
</tr>
<tr>
<td>Keyword</td>
<td><code>%{ key =&gt; value, key =&gt; value }</code></td>
</tr>
<tr>
<td>Enum</td>
<td>(can duplicate keys)</td>
</tr>
<tr>
<td>Binary</td>
<td><code>&lt; &lt; 1, 2 &gt;&gt; or &quot;abc&quot;</code></td>
</tr>
<tr>
<td></td>
<td>&quot;&quot; &quot;here doc&quot; &quot;&quot;</td>
</tr>
<tr>
<td></td>
<td>&quot;#(interpolated)&quot;</td>
</tr>
<tr>
<td></td>
<td><code>&lt; name:prop-prop-prop ... &gt;&gt;</code></td>
</tr>
<tr>
<td></td>
<td>binary, bits, bitstring, bytes, float, integer, utf8, utf16, utf32, size(n), signed/unsigned, big/little native</td>
</tr>
<tr>
<td>Truth</td>
<td>true, false, nil</td>
</tr>
<tr>
<td>Range</td>
<td><code>a..b</code></td>
</tr>
</tbody>
</table>

**Modules**

```elixir```
```
defmodule mod_name do
  @moduledoc "description"
  @doc "description"
  function/macro
end```

**Guard Clause**

Part of pattern match

```elixir```
```
when expr
  where operators in expr are limited to:
  ==, !=, ===, !==, >, <, <<=, >>=, or, and, not, !, +, -, *, /, in, is_atom, is_binary, is_bitstring, is_boolean, is_exception, is_float, is_function, is_integer, is_nil, is_list, is_number, is_pid, is_port, is_reference, is_tuple, abs(num), bit_size(bits), byte_size(bits), div(num,num), elem(tuple, n), float(term), head(list), length(list), node(), node(pid|ref|port), rem(num,num), round(num), self(), tl(list), trunc(num), tuple_size(tuple)
< and ++ (left side literal)
```excel

**Anonymous Functions**

```elixir```
```
f {fn parms [guard] -> body
  parms [guard] -> body
end
```

call with `func()`

Shortcut: `&(...) &1, &2 as parameters`

**Comprehensions**

```elixir```
```
for generator/filter [i | into: value] do: expr
```excel

**Named Functions**

```elixir```
```
def name{parms [guard] do
  expression
end
```excel

Generators are:

```
pattern <- list
```excel

With binaries as:

```elixir```
```
for << ch <- "hello" >>, do: expr
```excel

**Operators**

`===` `!==` and or not              (strict)

`==` `!=` `&&` `||` `!`             (relaxed)

`>`, `>=`, `<`, `<=`                 (float)

`div`, `rem`                       (integer)

`binary1 <> binary2`               (concat)

`list1 ++ list2`                   (concat)

`list1 -- list2`                   (set diff)

`a in enum`                       (membership)

`^` `term`                         (no reassign)

Copyright © 2013-2016 The Pragmatic Programmers, LLC. Free to use without modification for noncommercial applications. Content/design by Andy Hunt & Dave Thomas.
### Protocols

```elixir
defprotocol module.name do
  @moduledoc description
  @only [list of types] (optional)
  def name(parms) end
end
```

Allowed types:
- Any
- Atom
- BitString
- Function
- List
- Number
- PID
- Port
- Record
- Reference

### Control Flow

```elixir
if expr do
  exp
else
  exp
end
```

```elixir
with match <- exp,
  match <- exp,
  ...
  do: exp
```

executes all `exp` until a match fails (and is returned), or the `do:` is run.

### Metaprogramming

```elixir
defmacro macro_name(parms) do
  parms are quoted args
  return quoted code which
  is inserted at call site
end
```

```elixir
quote do: ...
  returns internal rep.
quote bind_quoted: [name: name] do: ...
```

```elixir
unquote do: ...
  only inside quote, injects
  code fragment without evaluation
```

### Processes

```elixir
pid = spawn(anon_function)  
pid = spawn(mod, func, args) 
(also spawn_link)
```

```elixir
receive do
  {sender, msg, ...} -> 
    send sender {ok, value} 
  after timeout ->  
    ...
end
```

### Regexp

```elixir
~r{pattern}opts
```

| f | match beg of ml string |
| g | use named groups       |
| i | case insensitive       |
| m | ^ and $ match each line in multiline |
| r | reluctant (not greedy) |
| s | . matches newline      |
| u | Unicode patterns       |
| x | ignore whitespace and comments |

### Sigils

```elixir
~r regex w/interpolation
~s string (no interpolation)
~S string (with interpolation)
~C character list (no interpolation)
~C character list (with interpolation)
~R regexp
```

### Maps

```elixir
%x{ key => value, key => value }
```

```elixir
value = map[key] (can return nil)
value = map.assoc(key) (can fail)
newmap = %{ oldmap | key => newval }
newmap = Map.put(oldmap, key, newval) 
Map.put_new/3 to add a key
```

### Pipelines

```elixir
expr |> f1 |> f2(a, b) |> f3(c)
```

(same as)

```elixir
f3(f2(f1(expr), a, b), c)
```

### Predefined Names

```elixir
__MODULE__
__FILE__
__DIR__
__ENV__
__CALLER__
```

(macros only)