We will define a small subset of the Elixir language and describe the *operational semantics*.

Warning - this is not exactly how Elixir works ... but it could have been.

**expressions**

The language is described using a BNF notation.

\[
\begin{align*}
  \langle \textit{atom} \rangle &::= :a | :b | :c | \\
  \langle \textit{variable} \rangle &::= x | y | z | \\
  \langle \textit{literal} \rangle &::= \langle \textit{atom} \rangle \\
  \langle \textit{expression} \rangle &::= \langle \textit{literal} \rangle | \langle \textit{variable} \rangle | \{ \langle \textit{expression} \rangle , \langle \textit{expression} \rangle \}
\end{align*}
\]

Examples: \{a, b\}, \{x, y\}, \{a, \{b, z\}\}

Simple expressions are also referred to as *terms*.

**patterns**

A *pattern* is a syntactical construct that uses almost the same syntax as terms.

\[
\begin{align*}
  \langle \textit{pattern} \rangle &::= \langle \textit{literal} \rangle \\
  &| \langle \textit{variable} \rangle \\
  &| _ \\
  &| \{ \langle \textit{pattern} \rangle , \langle \textit{pattern} \rangle \}
\end{align*}
\]

The _ symbol can be read as “don’t care”.

---

Johan Montelius

KTH

VT19
sequence

(match) ::= <pattern> '=' <expression>

(sequence) ::= <expression>
  | <match> ';' <sequence>

examples:
  • x = :a; {:b, x}
  • x = :a; y = {:b, x}; {:a, y}

evaluation

When we evaluate sequences, the result will be a structure.

Atoms = \{a, b, c,...\}

Structures = Atoms \cup \{\{s_1, s_2\}|s_i \in \text{Structures}\}

An evaluation can also result in \perp, called “bottoms”, this represents a failed evaluation.

injective mapping

For every atom \(a\), there is a corresponding structure \(s\).

We write \(a \equiv s\).

:foo \equiv \text{foo}
:gurka \equiv \text{gurka}

For every digit 1, 2, 3 (or I, II, III) there is a corresponding number 1, 2, 3.

Our language could have structures that do not have corresponding terms.

evaluation

A sequence is evaluated given an environment, written \(\sigma\) (sigma).

The environment holds a set of variable substitutions (bindings): \(v/s \in \sigma\), \(v\) is a variable and \(s\) is a structure.

An evaluation of a sequence \(e\) given an environment \(\sigma\) is written \(E_\sigma(e)\).

We write:

\[
\frac{\text{prerequisite}}{E_\sigma(\text{expression}) \rightarrow \text{result}}
\]

where result is a structure.
We have the following rules for evaluation of expressions:

Evaluation of an atom:
\[a \equiv s\]
\[E_\sigma(a) \rightarrow s\]

Evaluation of a variable:
\[v/s \in \sigma\]
\[E_\sigma(v) \rightarrow s\]

Evaluation of a compound structure:
\[E_\sigma(e_1) \rightarrow s_1\]
\[E_\sigma(e_2) \rightarrow s_2\]
\[E_\sigma(\{e_1, e_2\}) \rightarrow \{s_1, s_2\}\]

What if we have \(E_\sigma(v)\) and \(v/s \notin \sigma\)?

\[v/s \notin \sigma\]
\[E_\sigma(v) \rightarrow \bot\]

The result of evaluating a pattern matching is a an extended environment. We write:

\[P_\sigma(p, s) \rightarrow \theta\]

where \(\theta\) (theta) is the extended environment.

Match an atom:
\[a \equiv s\]
\[P_\sigma(a, s) \rightarrow \sigma\]

Match an unbound variable:
\[v/t \notin \sigma\]
\[P_\sigma(v, s) \rightarrow \{v/s\} \cup \sigma\]

Match a bound variable:
\[v/s \in \sigma\]
\[P_\sigma(v, s) \rightarrow \sigma\]

Match ignore:
\[P_\sigma(_, s) \rightarrow \sigma\]
What do we do with $P_\sigma(a, s)$ when $a \not\equiv s$?

$$\frac{a \not\equiv s}{P_\sigma(a, s) \rightarrow \text{fail}}$$

$$\frac{v/t \in \sigma \quad t \not\equiv s}{P_\sigma(v, s) \rightarrow \text{fail}}$$

A fail is not the same as $\bot$.

If the pattern is a compound pattern, the components of the pattern are matched to their corresponding substructures.

$$\frac{P_\sigma(p_1, s_1) \rightarrow \sigma' \\ P_\sigma'(p_2, s_2) \rightarrow \theta}{P_\sigma(\{p_1, p_2\}, \{s_1, s_2\}) \rightarrow \theta}$$

Note that the second part is evaluated in $\sigma'$.

Example: $P\{}(\{x, \{y, x\}\}, \{a, \{b, c\}\})$

Match a compound pattern with anything but a compound structure will fail.

Pattern matching can fail.

fail is different from $\bot$

We will use failing to guide the program execution, more on this later.
A new **scope** is created by removing variable bindings from an environment.

\[
\sigma' = \sigma \setminus \{v/t \mid v/t \in \sigma \land v \in p\}
\]

\[
S(\sigma, p) \rightarrow \sigma'
\]

A sequence is evaluated one pattern matching expression after the other.

\[
E_\sigma(e) \rightarrow t \quad S(\sigma, p) \rightarrow \sigma' \quad P_{\sigma'}(p, t) \rightarrow \theta \quad E_\theta(\text{sequence}) \rightarrow s
\]

\[
E_\sigma(p = e; \text{sequence}) \rightarrow s
\]

**Erlang and Elixir differ in how this rule is defined.**

---

**Where are we now**

We have defined the semantics of a programming language (not a complete language) by defining how expressions are evaluated.

Important topics:
- set of data structures: atoms and compound structures
- environment: that binds variables to data structures
- expressions: term expressions, pattern matching expressions and sequences
- evaluation: from expressions to data structures \( E_\sigma(e) \rightarrow s \)

**Why do we do this?**
What is missing:
- evaluation of case (and if expressions)
- evaluation of function applications

```plaintext
case x do
  :a -> :foo
  :b -> :bar
end
```

### case expression

\[
\langle \text{expression} \rangle ::= \langle \text{case expression} \rangle | ...
\]

\[
\langle \text{case expression} \rangle ::= \text{case} \ <\text{expression}> \ \text{do} \ <\text{clauses}> \ \text{end}
\]

\[
\langle \text{clauses} \rangle ::= \langle \text{clause} \rangle | \langle \text{clause} \rangle ; \langle \text{clauses} \rangle
\]

\[
\langle \text{clause} \rangle ::= \langle \text{pattern} \rangle \ \text{->} \ <\text{sequence}>\]

### evaluation of case expression

\[
E_{\sigma}(e) \rightarrow t \quad C_{\sigma}(t, \text{clauses}) \rightarrow s
\]

\[
E_{\sigma}(\text{case e do clauses end}) \rightarrow s
\]

\[C_{\sigma}(s, \text{clauses}) \text{ will select one of the clauses based on the patterns of the clauses and then continue the evaluation of the sequence of the selected clause.}\]
selection of a clause

\[
S(\sigma, p) \rightarrow \sigma' \quad P\sigma'(p, s) \rightarrow \theta \quad \theta \neq \text{fail} \quad E\theta(\text{sequence}) \rightarrow s
\]
\[
C\sigma(s, p \rightarrow \text{sequence}; \text{clauses}) \rightarrow s
\]
\[
S(\sigma, p) \rightarrow \sigma' \quad P\sigma'(p, s) \rightarrow \text{fail} \quad C\sigma(s, \text{clauses}) \rightarrow s
\]
\[
C\sigma(s, p \rightarrow \text{sequence}) \rightarrow \perp
\]

element

\[
E\{x/{a, b}\}(\text{case } x \text{ of } :a \rightarrow :a; \{_, y\} \rightarrow y \text{ end}) \rightarrow
\]
\[
E\{X/{a, b}\}(x) \rightarrow \{a, b\}
\]
\[
C\{X/{a, b}\}(\{a, b\}, :a \rightarrow :a; \{_, y\} \rightarrow y) \rightarrow
\]
\[
P\{x/{a, b}\}(\{a, b\}) \rightarrow \text{fail}
\]
\[
C\{x/{a, b}\}(\{a, b\}, \{_, y\} \rightarrow y) \rightarrow
\]
\[
P\{x/{a, b}\}(\{_, y\}, \{a, b\}) \rightarrow \{y/b, x/{a, b}\}
\]
\[
E\{y/b, x/{a, b}\}(y) \rightarrow
\]
\[
b
\]

free variables

Are all syntactical correct sequences also valid sequences?

A sequence must not contain any free variables.

A free variable in a <sequence> is bound by the pattern matching expressions in the sequence <patter> = <expression>, <sequence> if the variable occurs in the <pattern>.

A free variable in a <sequence> is bound by the pattern matching expressions in the clause <pattern> -> <sequence> if the variable occurs in the <pattern>.

\[
x = :a; \{y, z\} = \{x,:b\}; \{x,y,z\}
\]
\[
\{y, z\} = \{x,:b\}; \{x,y,z\}
\]
\[
x = \{:a,:b\}; \text{case } x \text{ do } {:a,z} \rightarrow z \text{ end}
\]
A lot of work for something that simple - why bother, it could not have been done differently.

```
x = {:a,:b};
y = case x do
   {:a, z} -> {:c, z}
   end;
{y, z}
```

*This is not allowed in our language, z in \(y, z\) is a free variable. However .... is allowed in Erlang and was until changed allowed in Elixir (fixed in v1.5).*

Handle lambda expressions, closures and function application.