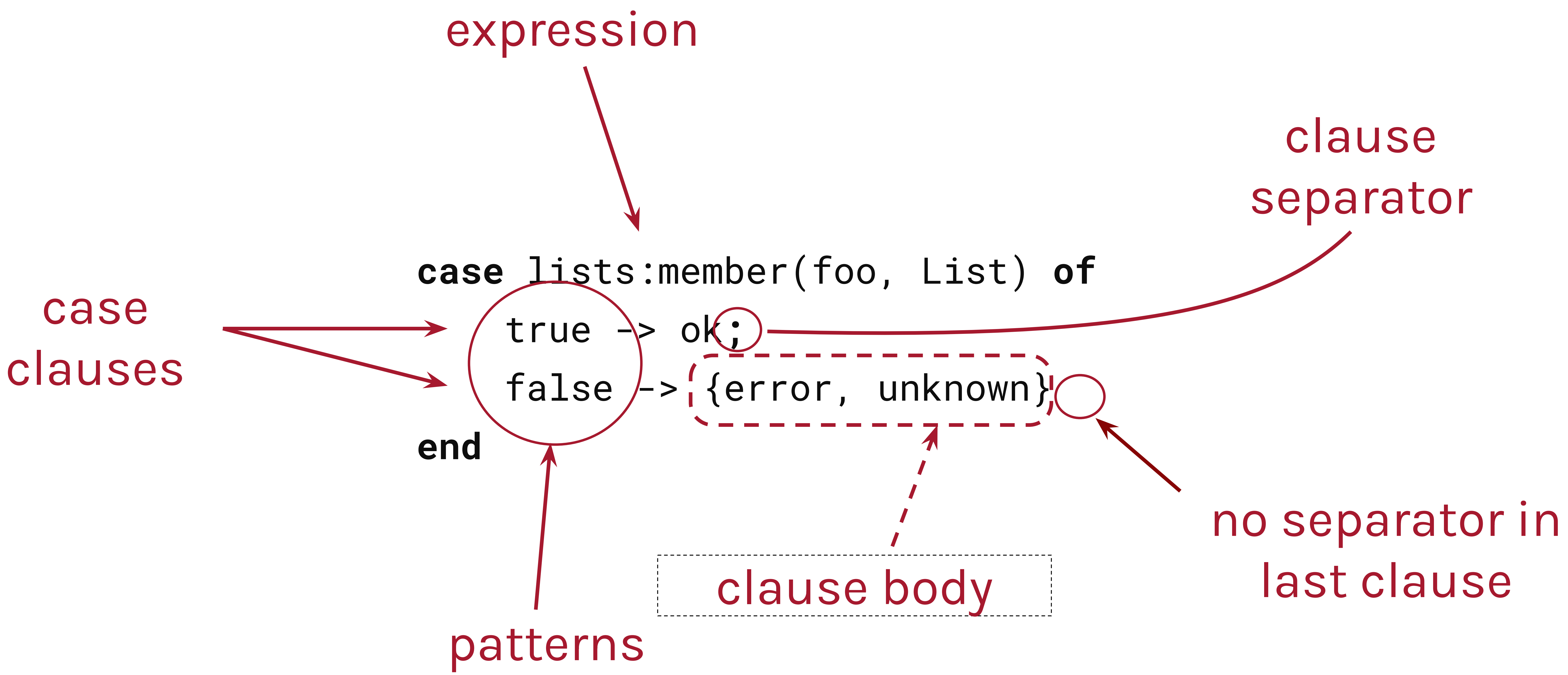
Sequential Erlang

Overview: Sequential Erlang

- ▶ Conditional Evaluation
  - ○ Case Statements
  - ○ Guards
  - ○ If Statements
- ▶ Recursion

Conditional Evaluation: case



```
case lists:member(foo, List) of
    true -> ok;
    false -> {error, unknown}
end
```

expression

clause separator

case clauses

clause body

no separator in last clause

patterns

Conditional Evaluation: case

```
case <expression> of
    Pattern1 ->
        <expression 1>,
        <expression 2>,
        ...
        <expression N>;
   Pattern2 ->
        <expression 1>,
        <expression 2>,
        ...
        <expression N>;
   _ ->
        <expression 1>,
        ...
        <expression N>
end
```

- One branch should always succeed
- Using an unbound variable or '_' ensures that the clause will always match
- The _ clause is not mandatory
- An exception is raised if no clause matches
- Returns the value of the last executed expression

Defensive Programming

```
convert(Day) ->
  case Day of
      monday    -> 1;
      tuesday   -> 2;
      wednesday -> 3;
      thursday  -> 4;
      friday    -> 5;
      saturday  -> 6;
      sunday    -> 7
  end.
```

- Defensive programming: program in the convert function for the error case or …
- … let it fail here by deleting the **_Other** clause.
- This will raise an exception
- The caller will have to handle the error that they have caused.

Guards

```
factorial(N) when N > 0 ->
    N * factorial(N - 1);
factorial(0) -> 1.
```

This is NOT the same as...

```
factorial(0) -> 1;
factorial(N) ->
    N * factorial(N - 1).
```

- The reserved word **when** introduces a guard
- Fully guarded clauses can be re-ordered
- Guards can be used in function heads, case clauses, receive and if expressions.

Guards: examples

```
number(Num) when is_integer(Num) -> integer;
number(Num) when is_float(Num) -> float;
number(_Other) -> false.
```

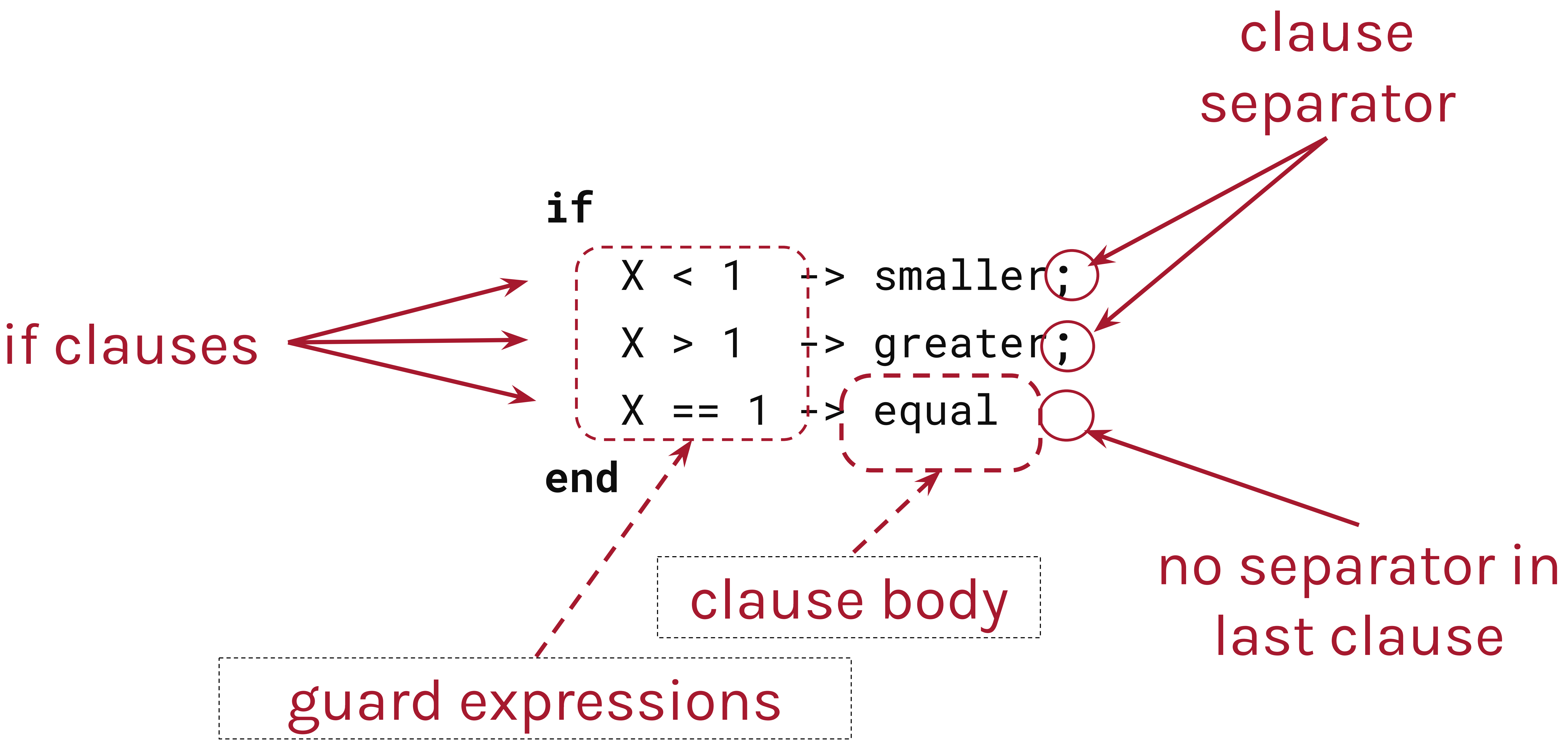
- is_number(X), is_integer(X), is_float(X), is_atom(X), is_pid(X), is_tuple(X), is_list(X)
  - **X is the specified datatype**
- length(List) == Int, tuple_size(Tuple) == Size, X > Y + Z
  - Some BIFs and mathematical applications can be applied in guards
- X == Y   X /= Y   X =:= Y   X =/= Y
  - X is (not) equal to Y, X is exactly (not) equal to Y (1==1.0 ✓, 1=:=1.0 ×)
- X =< Y   X >= Y
  - **Note: not <= or =>**

Guards

```
legal_age(Age) when is_integer(Age), Age >= 18 -> true;
legal_age(Age) when is_integer(Age), Age < 18  -> false.
```

- All variables in guards have to be bound
- Guards have to be free of side effects
- If all the guards have to succeed, use , to separate them
- If one guard has to succeed, use ; to separate them
- There are restrictions on BIFs and expressions in guards
  - See the Erlang reference manual for complete details

Conditional Evaluation: if



```
if
   X < 1  -> smaller;
   X > 1  -> greater;
   X == 1 -> equal
end
```

clause separator

if clauses

clause body

guard expressions

no separator in last clause

Conditional Evaluation: if

```
if Guard1 ->
        <expression 1>,
        <expression 2>,
        ...
        <expression N>;
   Guard2 ->
        <expression 1>,
        <expression 2>,
        ...
        <expression N>;
   ...
   true ->
        <expression 1>,
        ...
        <expression N>
end
```

- One branch must always succeed
- By using **true** as the last guard, we ensure that a clause will always succeed
- The **true** guard is not mandatory
- An exception is raised if no clause succeeds
- Returns the value of the last executed expression

General Switch

```
if f(Args) -> ok;
   true    -> error
end
```

✗

```
case f(Args) of
    true -> ok;
    false -> error
end
```

✓

- The if construct fails because it involves a user-defined function, which are forbidden in guards
- The case construct succeeds because it accepts user-defined functions.

Recursion: traversing lists

```
average(X) -> sum(X) / len(X).

sum([H|T]) -> H + sum(T);
sum([]) -> 0.

len([_|T]) -> 1 + len(T);
len([]) -> 0.
```

- Note the pattern of recursion is the same in both cases
- Taking a list and evaluating an element is a very common pattern

Recursion: self-describing code

```
sum([]) -> 0;

sum([H|T]) -> H + sum(T).
```

- You can read the programs as an executable description:
- "The sum of an empty list is 0."
- "The sum of a non-empty list is the head of the list added to the sum of the tail"

Recursion: traversing lists

```
printAll([]) ->
    io:format("~n", []);

printAll([X|Xs]) ->
    io:format("~p ", [X]),
    printAll(Xs).
```

- Here we're traversing the list imperatively:
- "If there are no more elements to process, stop"
- "If there are further elements, process the head, and then call the function recursively on the tail."

Recursion: traversing lists

```
printAll(Ys) ->
    case Ys of
      [] ->
        io:format("~n", []);
      [X|Xs] ->
        io:format("~p ", [X]),
        printAll(Xs)
    end.
```

- Same function again: shows the loop clearly. The call to **printAll(Xs)** is like a **jump** back to the top of the loop.
- This is a **tail recursive** function: the only recursive calls come at the end of the bodies of the clauses.

Recursion: more patterns

```
double([H|T])-> [2*H|double(T)];
double([]) -> [].

member(H, [H|_]) -> true;
member(H, [_|T]) -> member(H,T);
member(_, [])    -> false.

even([H|T]) when H rem 2 == 0 ->
    [H|even(T)];
even([_|T]) ->
    even(T);
even([]) ->
    [].
```

- **double/1** maps elements in a list and returns a new list
- **member/2** is a predicate looking for an element in a list
- **even/1** filters a list of integers and returns the subset of even numbers
- The function **member/2** is the only one which is tail recursive

Recursion: accumulators

```
average(X) -> average(X, 0, 0).

average([H|T], Length, Sum) ->
  average(T, Length+1, Sum+H);
average([], Length, Sum) ->
  Sum/Length.
```

- Only traverses the list once.
- Executes in constant space (tail recursive)
- **Length** and **Sum** play the role of accumulators
- **average([])** is not defined
- Evaluating **average([])** would cause a run time error.

Summary: Sequential Erlang

- Conditional Evaluation
- Guards
- Recursion