Exceptions

Failures and Results

Depending on the computation we want to specify, we might need to program operations that **fail**, e.g., when some value is outside some interval, like in a division with 0 as denominator or in an array lookup with an index that it outside its boundaries.

Besides being able to **express** these situations in programs, we should (must) also find ways to **structure** them, such that other operations that depend on the results of the failing one are aware of this possibility — and the static checks of the language can aid us dealing with them.

We already saw one solution to this problem with monadic **`Result`** types, which encapsulate and structure the passage/access of successful and failing outcomes.

However, while `Results` help making all possible errors explicit, they can become cumbersome to compose when nesting operations (e.g., when we compose one or more operations that do not handle the failure directly but just “forward” them to their caller). Moreover, `Results` force the developer to always “unwrap” their value to act on them, determining a larger (and more difficult to peruse) codebase.

Alternatives to Result types

Of course, there are (usually less refined/elegant) alternatives to `Results`

One is defining some **exceptional values** that the operation use to signal the failure to the caller, e.g., we might have a naïve implementation of the division that uses 0 to indicate a failure. Of course, this method is quite limited, since, e.g., we conflate legit and failing calls (e.g., 0/1 and 1/0).

Another way is through the **inversion of control** between the caller and the callee, where a function that might fail asks the caller to pass, besides its ordinary inputs, a function that it calls in case of failure. While this might work in principle (only in languages that support functions as operation arguments), it makes reasoning on the flow of programs more complex and the usage of operations more obscure (we might need to know what the operation does with our error-handling function to write them correctly).

Exceptions

Exceptions are an alternative to (and much more popular than) `Results`.

The idea behind exceptions it that an exceptional condition (a rare failure occurrence) causes a direct **transfer of control** to an **exception handler** defined at some point in the call stack of the program. This can be in the operation calling the failing one, in the one above the former or even completely absent, in which case (if done on purpose) it means that there is nothing the programmer can do to recover from the exception and the only solution is to abort the program.

For example, for the division operator, we can call it an (unrecoverable) exception the fact that the system runs out of memory, while failing with a denominator equal to 0 is its characterising behaviour.

Exception handling

C or Rust do not provide direct support to exception handling and users resort to one of the mentioned alternatives — e.g., Rust mainly relies on “smart” operators for `Result` types that minimise the length of the codebase.

Other languages, like Java, provide exception-handling constructs that lexically bind **exception handlers** to blocks of code, which they replace in case they “catch” an exception.

In Java, exceptions are so embedded in the language that calling `/` with 1 and 0 as arguments, i.e., `1/0`, causes the expression to raise a

```
java.lang.ArithmeticException: / by zero
```

Exception handling

To handle exceptions, Java (and similar languages) provides the **try**-**catch** construct, e.g.,

```
try {
 System.out.println( “Let’s try to divide by zero” );
 double x = 1 / 0;
} catch ( ArithmeticException exception ){
 System.err.println( “You shall not divide by zero!” );
}
```

Where the **try** block encloses the code that might raise some exception and the **catch** block encloses the behaviour that we need to execute if we catch some (specific) exception.

Notably, the code in the **try** block does not execute “atomically”. Indeed, the code in the **catch** block replaces the code in the **try** only from the instruction that raises (**throws**, in Java-speak) the exception. In the example, the output (on the standard I/O) of our program prints the first string, “`Let’s try …`”, followed by the second one, “`You shall not …`”.

Exception handling

To handle exceptions, Java (and similar languages) provides the **try-catch** construct, e.g.,

```
try {
 System.out.println( “Let’s try to divide by zero” );
 double x = 1 / 0;
} catch ( ArithmeticException exception ){
 System.err.println( “You shall not divide by zero!” );
}
```

Note that the **catch** block accepts an argument: the exception it **expects to intercept** (and bind). In general, to structure the handling of exceptions, `catch` clauses specify the exceptions they intercept via names, which languages frequently conflate into symbols of their type system.

In Java, for instance, this mix enhances the flexibility of the **try-catch construct via subtyping**: all raisable and catchable values are subtypes of the special type `Throwable`—named after the Java exception-rising statement **throw**. In our example, the catch block declares it expects to intercept exceptions of the (subtype) ArithmeticException.

Exceptions and Subtyping

```
class MyException extends Throwable { … }

try {
 throw new MyException();
} catch ( MyException exception ){
 System.err.println( “Caught an exception!” );
}
```

In Java, developers rarely define direct subtypes of the `Throwable` type, but rather use subtypes of the two `Throwable`-subtypes `Error` and `Exception`.

Exceptions and Subtyping: Explicit vs Implicit Exceptions

The distinction between `Errors` and `Exceptions` is that the Java type system forces the developer to explicitly handle any `Throwable` subtype, save for Errors and a special subtype of `Exception`, called `RuntimeException`.

$$\begin{tikzcd} & \texttt{Error} & & \\ \texttt{Throwable} \arrow[ur, phantom, ":>"] \arrow[dr, phantom, ":>"] & & \texttt{RuntimeException} \arrow[r, phantom, ":>"] & \ldots \\ & \texttt{Exception} \arrow[ur, phantom, ":>"] \arrow[dr, phantom, ":>"] & & \\ & & \ldots & \end{tikzcd}$$

`Errors` and RuntimeExceptions respectively represent runtime-level and application-level **unrecoverable failures** that should abort the execution of the program, so the programmer should handle them only if they know (they exist and) how to recover from them.

Thus, in Java, exceptions are... not really that exceptional (in the sense of the definition of "rare failure occurrences") and are the predominant construct the language provides to handle failures.

Explicit Exceptions

Focussing on `Exceptions`, Java forces developers to explicitly handle them in two possible ways. Either they declare that an operation **throws** an `Exception` or they must handle the thrown exception within the operation.

```
void o() throws MyException{
 throw new MyException();
}
```

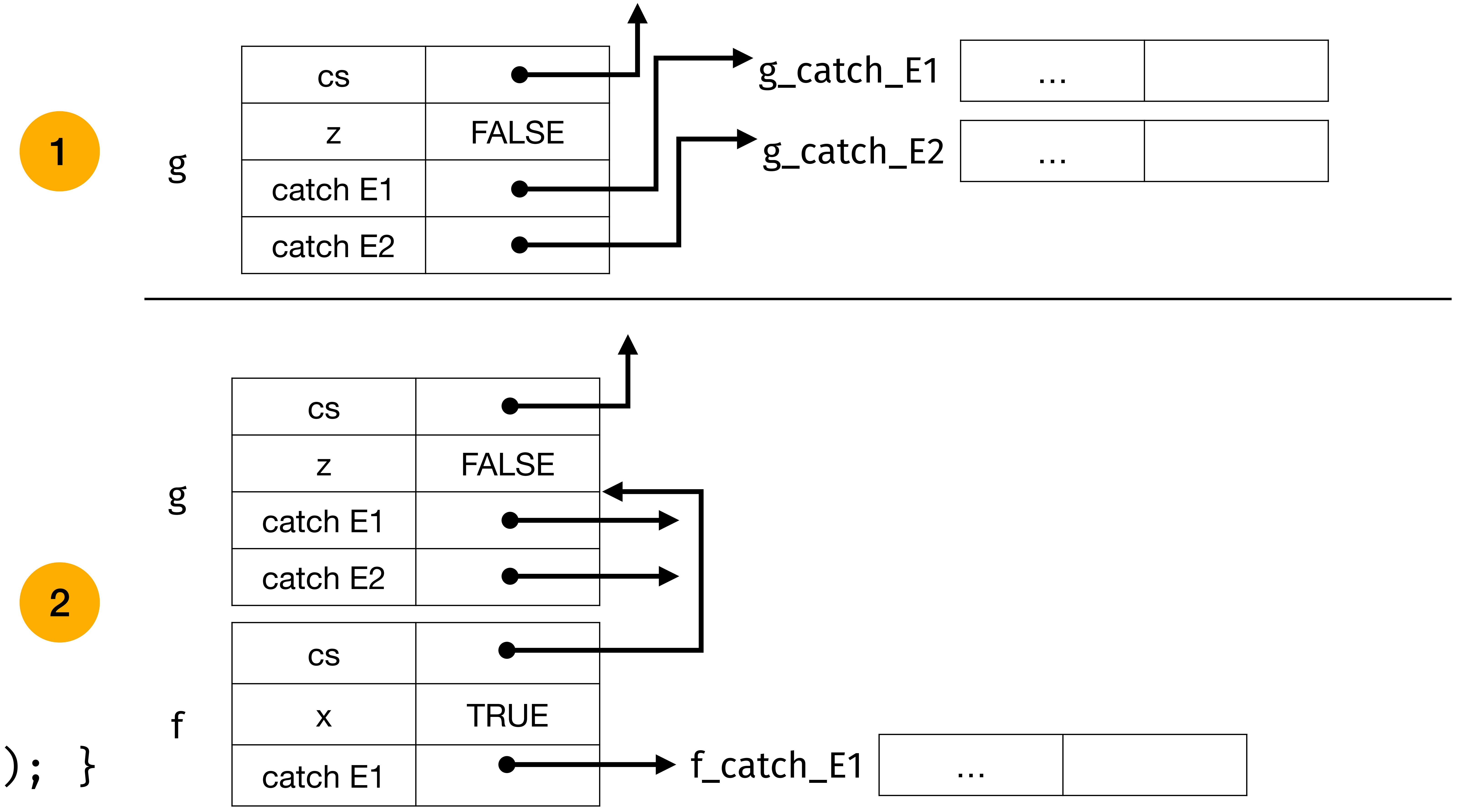
```
void o(){
 try {
  throw new MyException();
 } catch ( MyException e ){
  …
 }
}
```

Implementing try-catch behaviours

```
void e() throws E1 {
  bool x = true;
  throw new E1();
}

void f(bool x) throws E1, E2 {
 if( x ){
  try { e(); bool y = false; }
  catch( E1 e ){
    throw new E2();
   }
 } else {
    e();
} }

void g() {
 try { bool z = false; f( true ); }
 catch( E1 e ){}
 catch( E2 e ){}
}
```

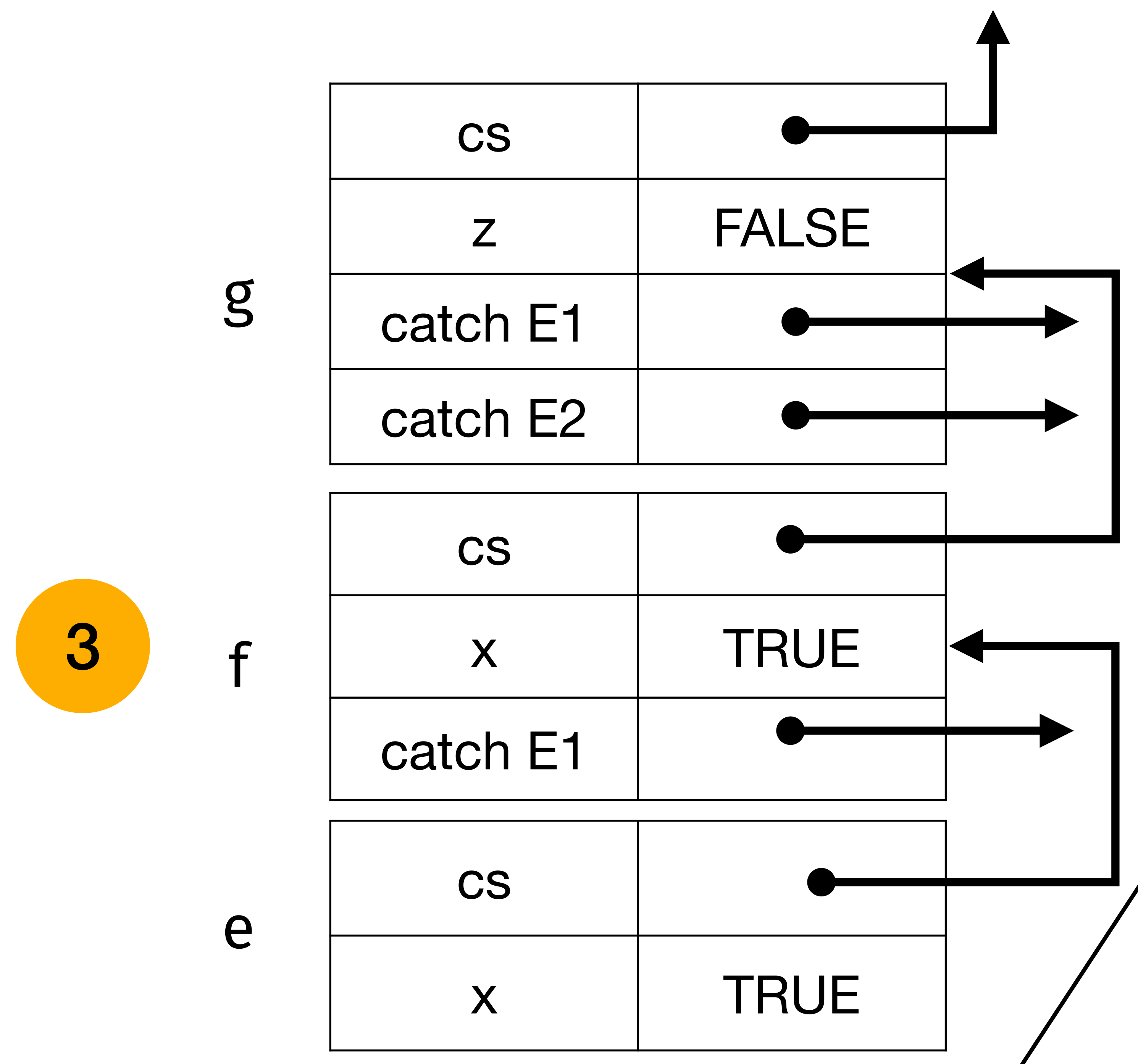


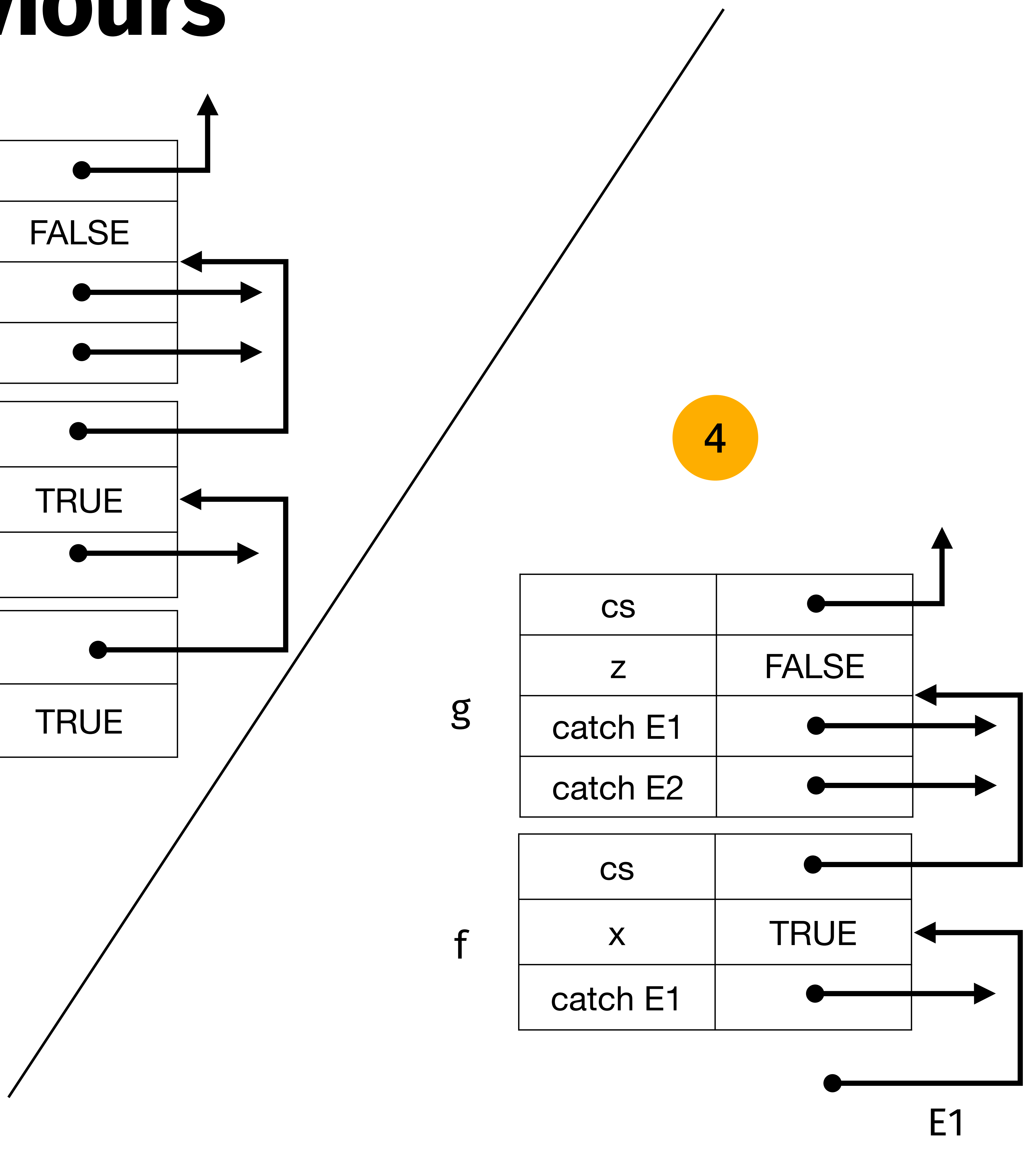
Implementing try-catch behaviours

```
void e() throws E1 {
  bool x = true;
  throw new E1();
}

void f(bool x) throws E1, E2 {
 if( x ){
  try { e(); bool y = false; }
  catch( E1 e ){
    throw new E2();
   }
 } else {
    f();
} }

void g() {
 try { bool z = false; f( true ); }
 catch( E1 e ){}
 catch( E2 e ){}
}
```



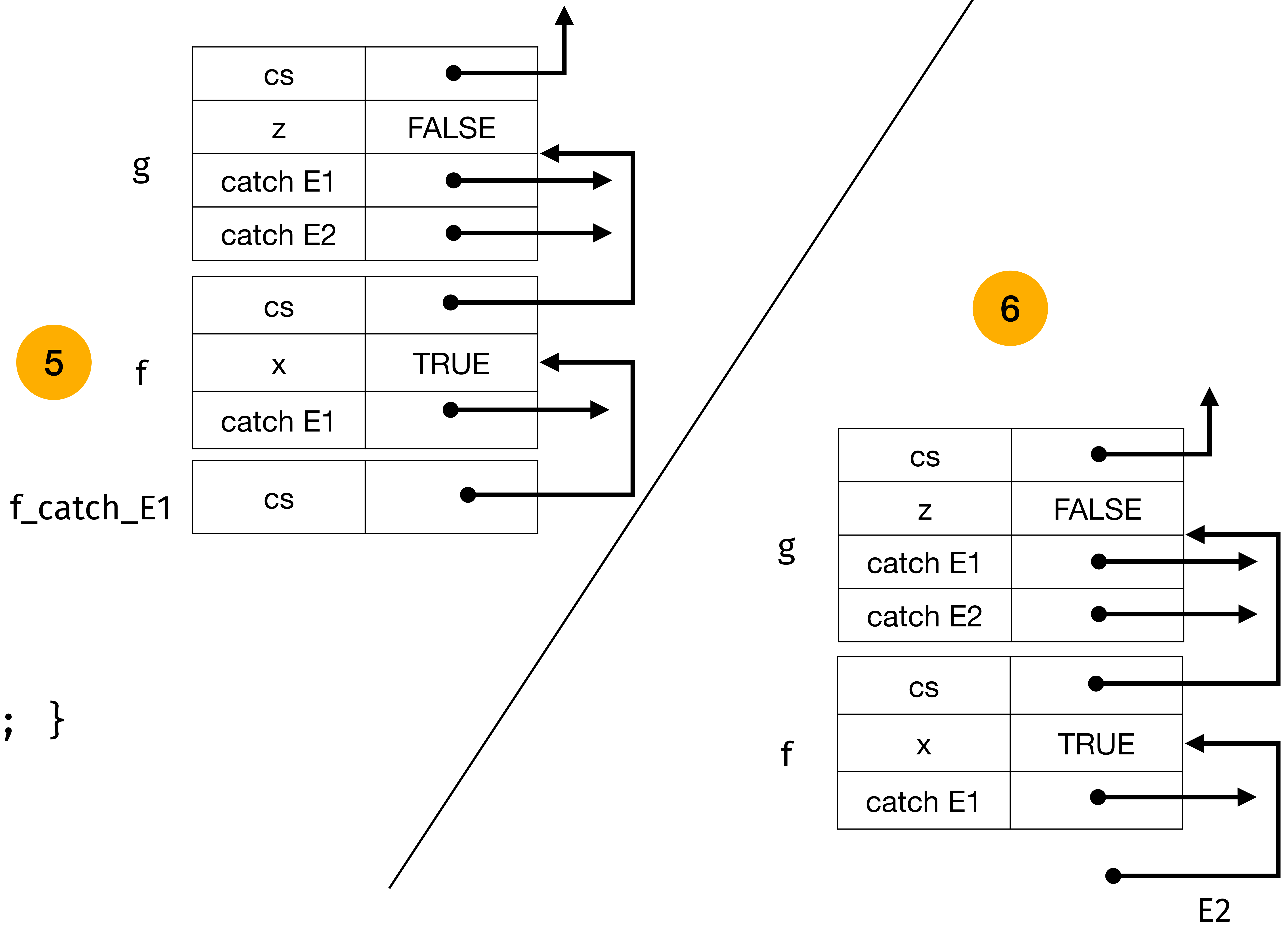


Implementing try-catch behaviours

```
void e() throws E1 {
  bool x = true;
  throw new E1();
}

void f(bool x) throws E1, E2 {
 if( x ){
  try { e(); bool y = false; }
  catch( E1 e ){
    throw new E2();
   }
 } else {
   f();
} }

void g() {
 try { bool z = false; f( true ); }
 catch( E1 e ){}
 catch( E2 e ){}
}
```

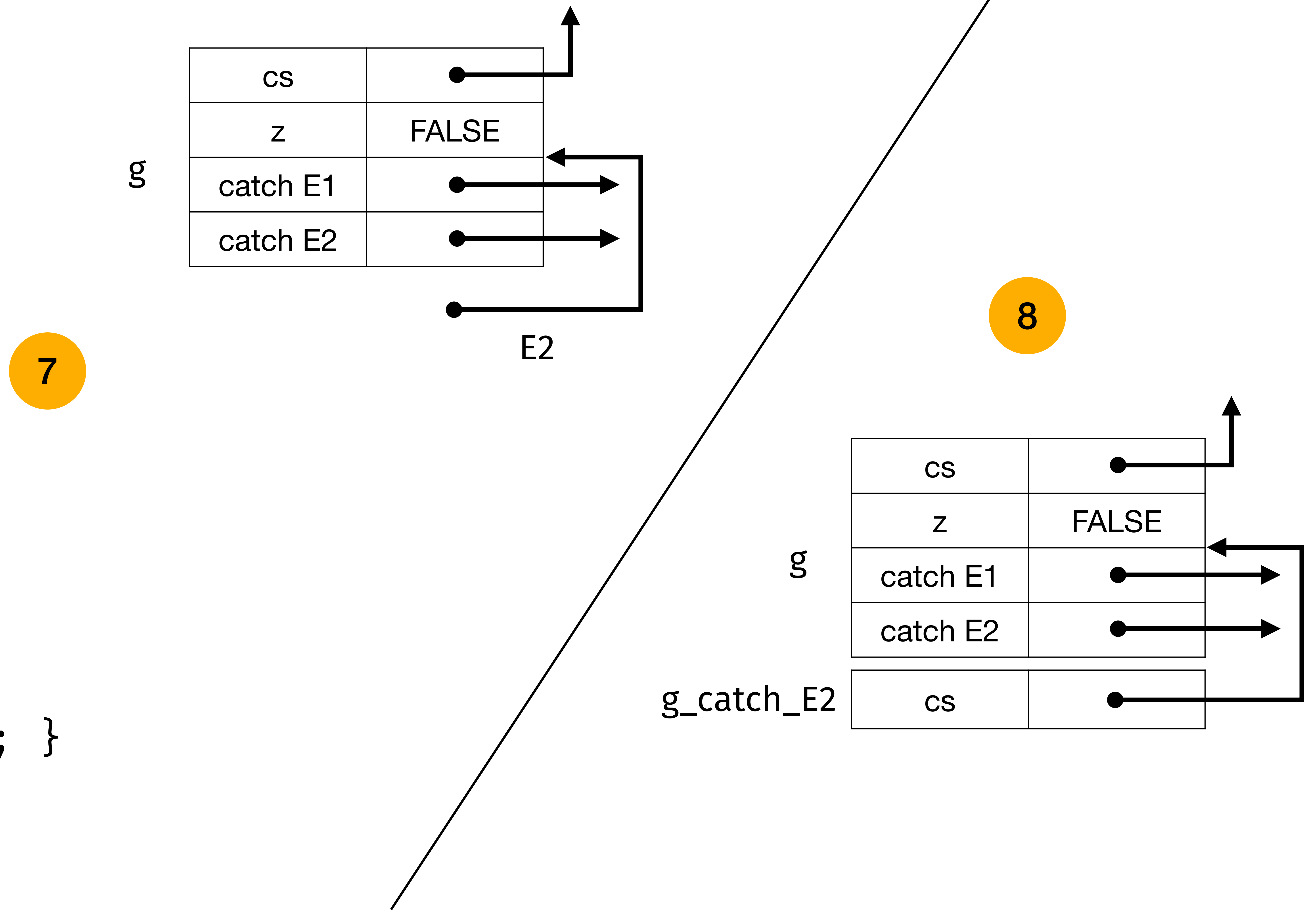


Implementing try-catch behaviours

```
void e() throws E1 {
  bool x = true;
  throw new E1();
}

void f(bool x) throws E1, E2 {
 if( x ){
  try { e(); bool y = false; }
  catch( E1 e ){
    throw new E2();
   }
 } else {
    f();
} }

void g() {
 try { bool z = false; f( true ); }
 catch( E1 e ){}
 catch( E2 e ){}
}
```


7
g
cs
z
FALSE
catch E1
catch E2
E2
8
g
cs
z
FALSE
catch E1
catch E2
g_catch_E2
cs