The Truth of a Procedure

Lisa Lippincott

Meeting C++, November 2018

Why don't we routinely write down the reasoning behind our programs in a formal way, and have computers check it?

The mathematical tools we use for proofs present a poor user interface for procedural programming.

Logic

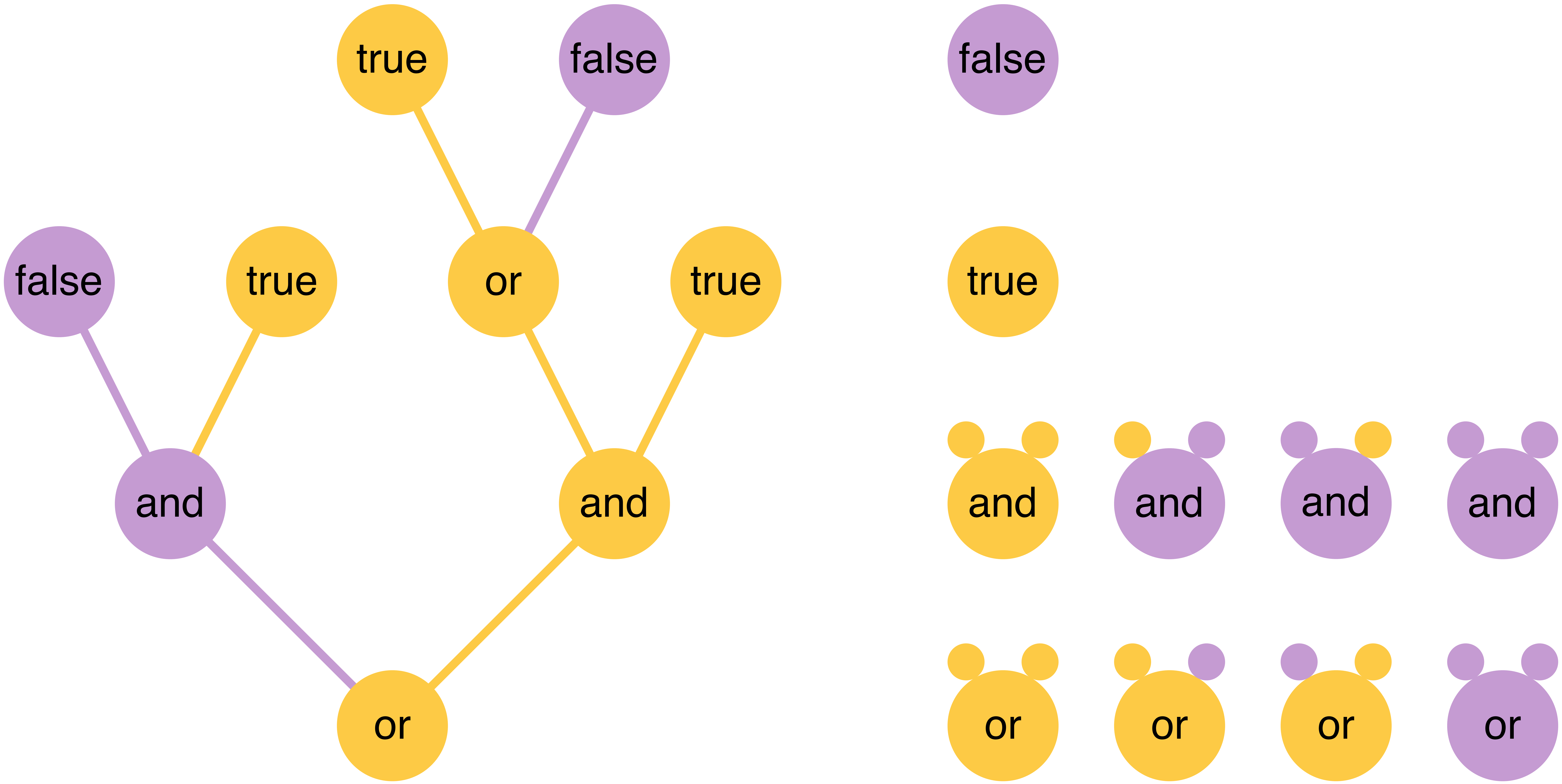
Procedural Logic

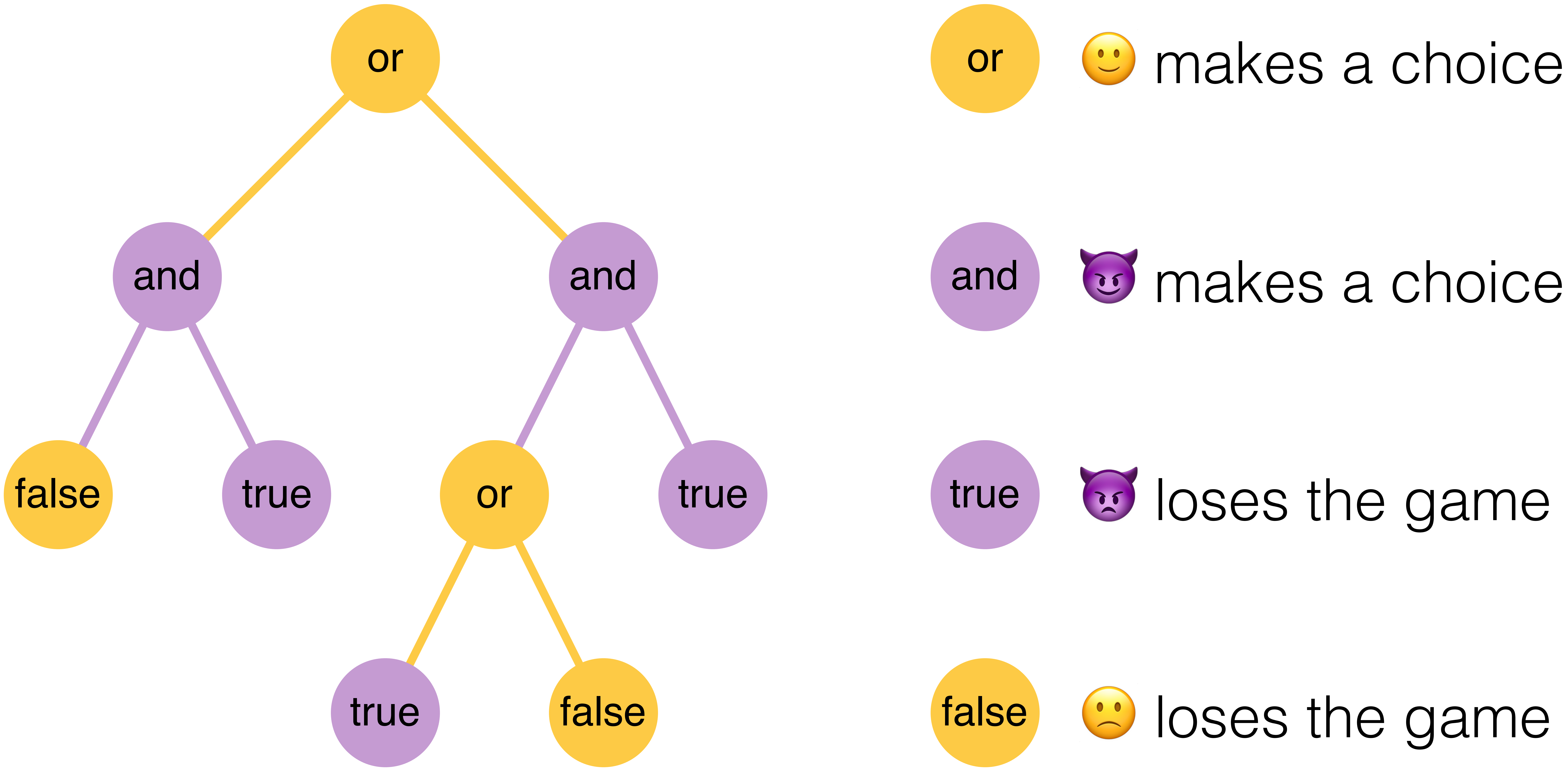
A procedure is an embodied algorithm, conceived as a scheme by which events may be arranged in time, space, possibility and causality.

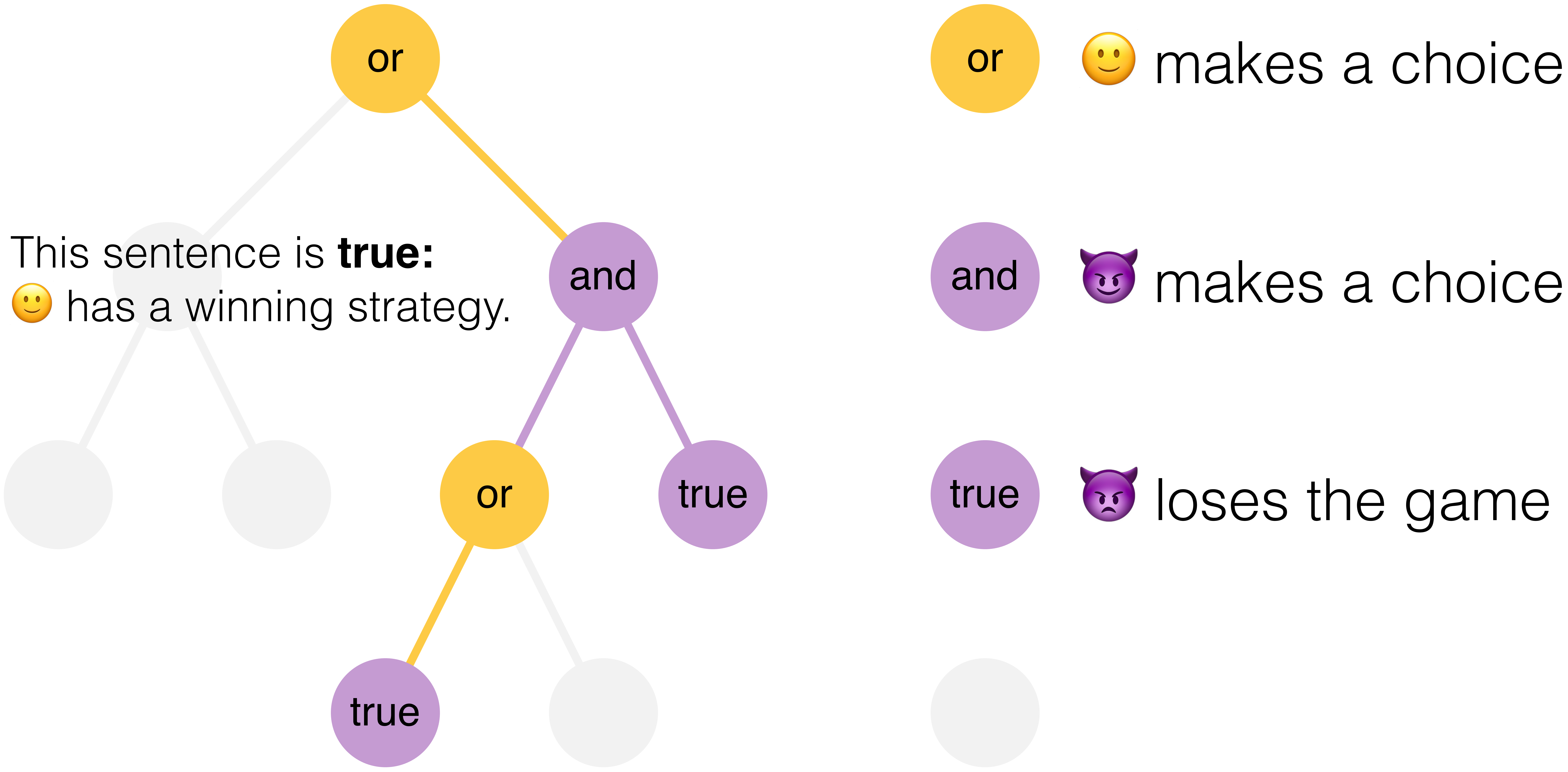
Procedures are sentences.

A sentence is a statement about the world, which may either be in agreement with the world (“true”) or be in disagreement with the world (“false”).

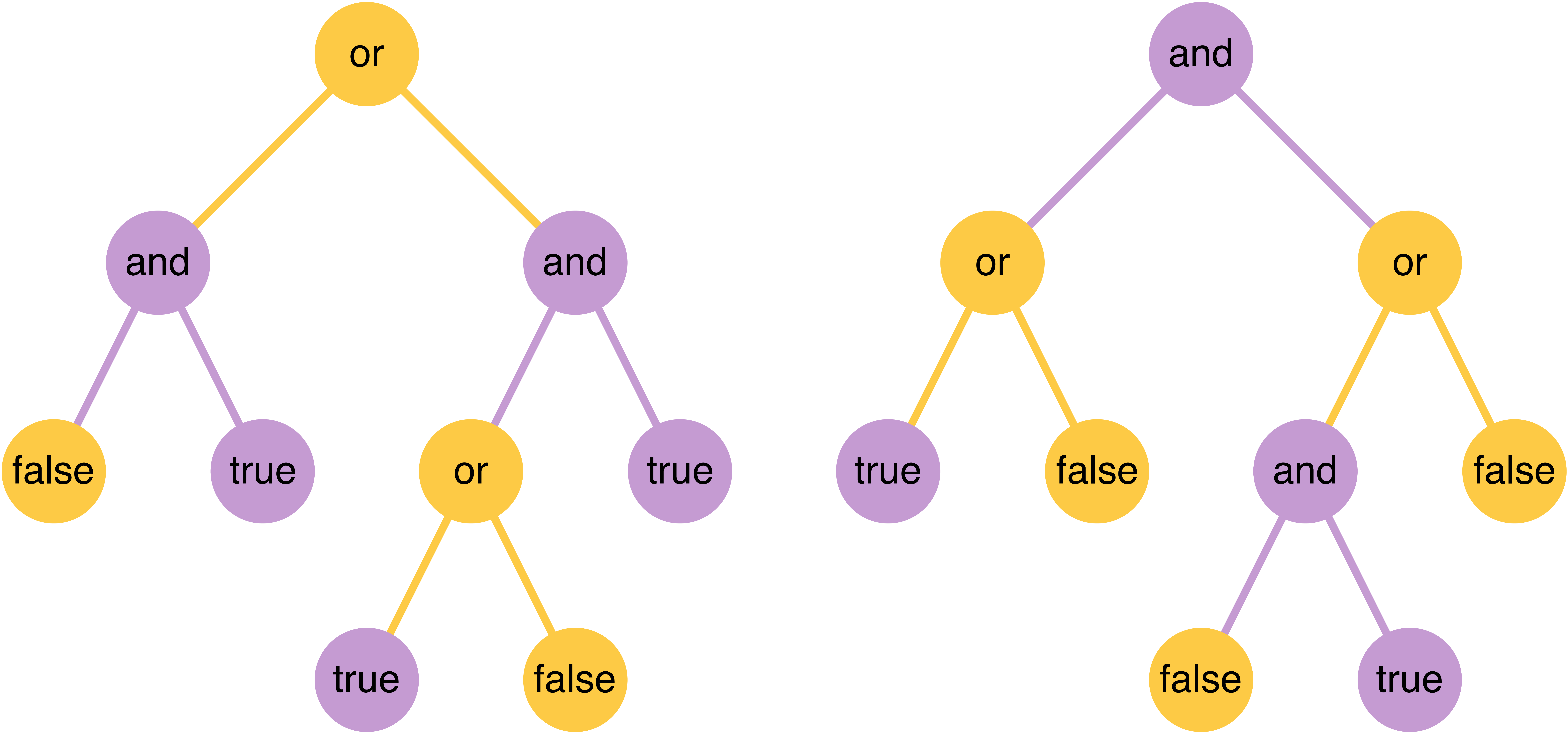
(false and true) or ((true or false) and true) —— Sentence


true
false
false
true
or
true
false
true
and
and
and
and
and
and
or
or
or
or
or

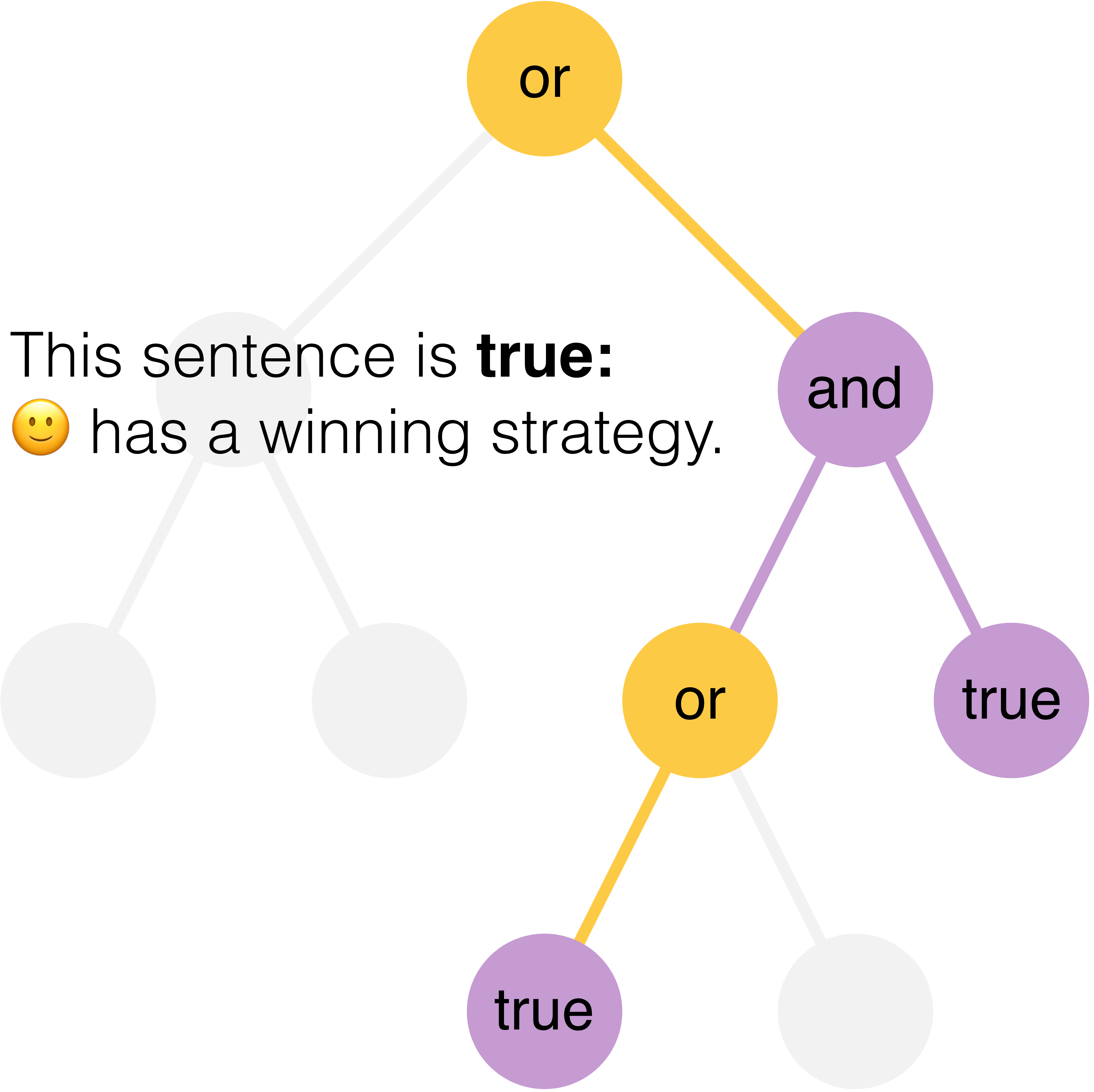

or
and
and
false
true
or
true
true
false
or 🙂 makes a choice
and 😈 makes a choice
true 👿 loses the game
false 🙁 loses the game



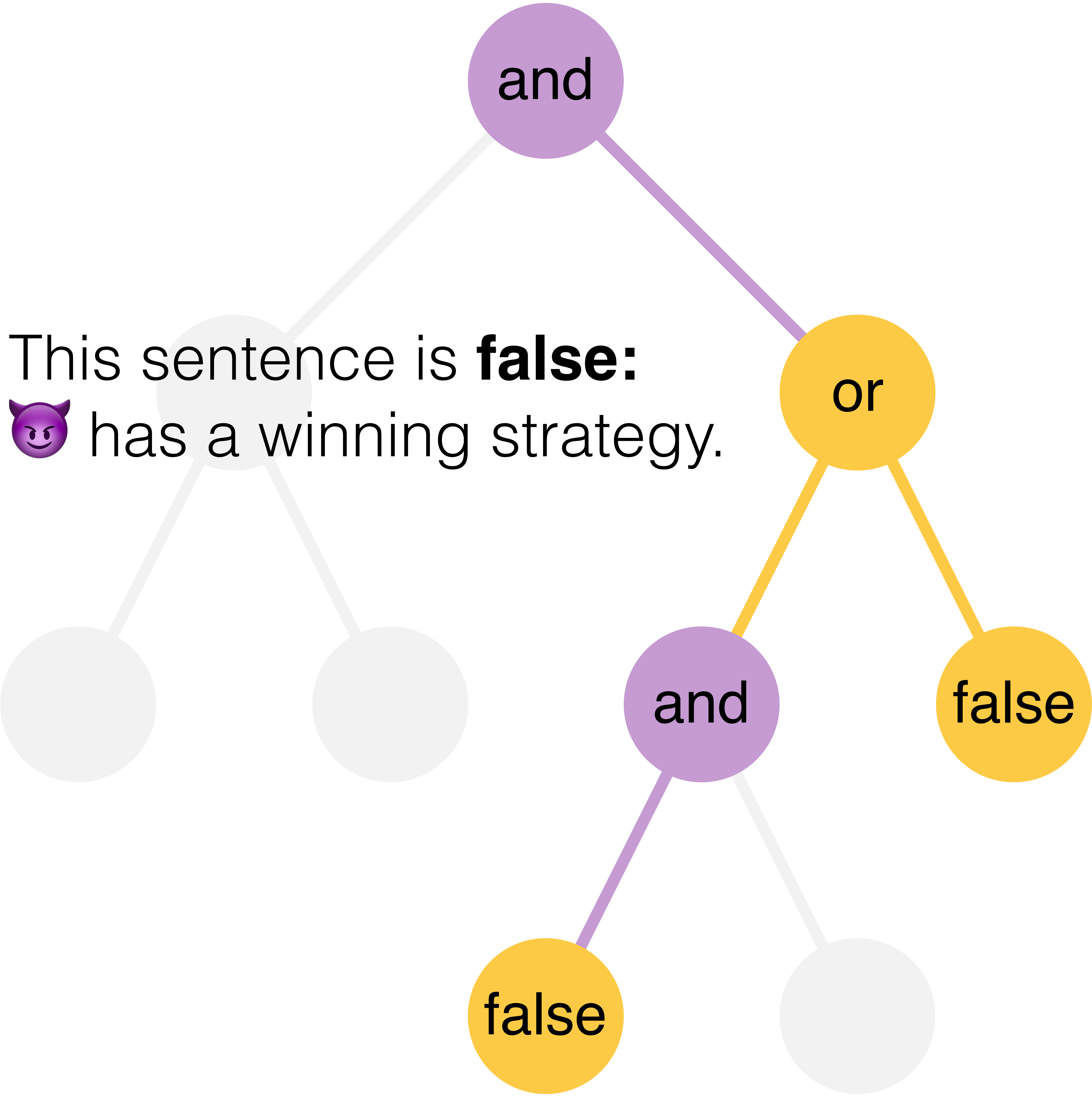
or
and
or
true
true
This sentence is **true:**
🙂 has a winning strategy.
or
🙂 makes a choice
and
😈 makes a choice
true
👿 loses the game


or
and
and
false
true
or
true
true
false
and
or
or
true
false
and
false
false
true

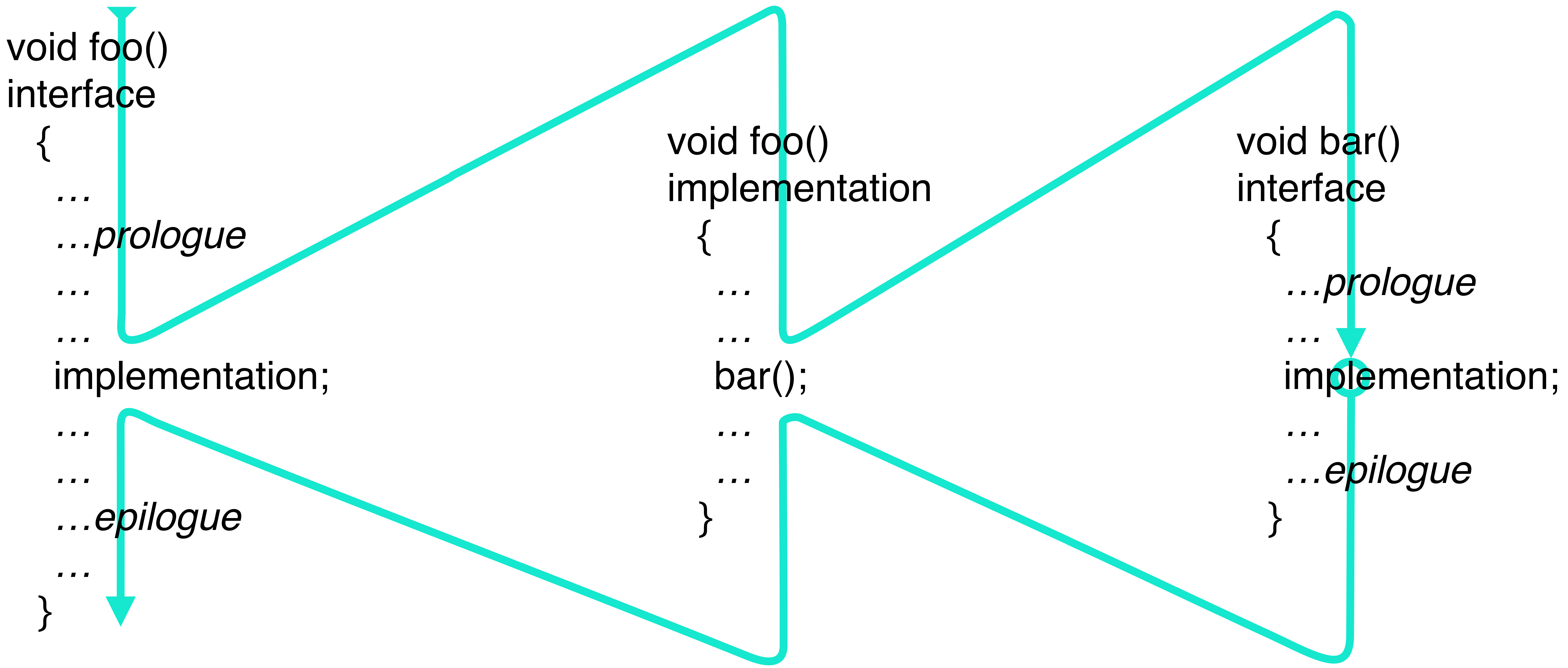
This sentence is **true:**
🙂 has a winning strategy.



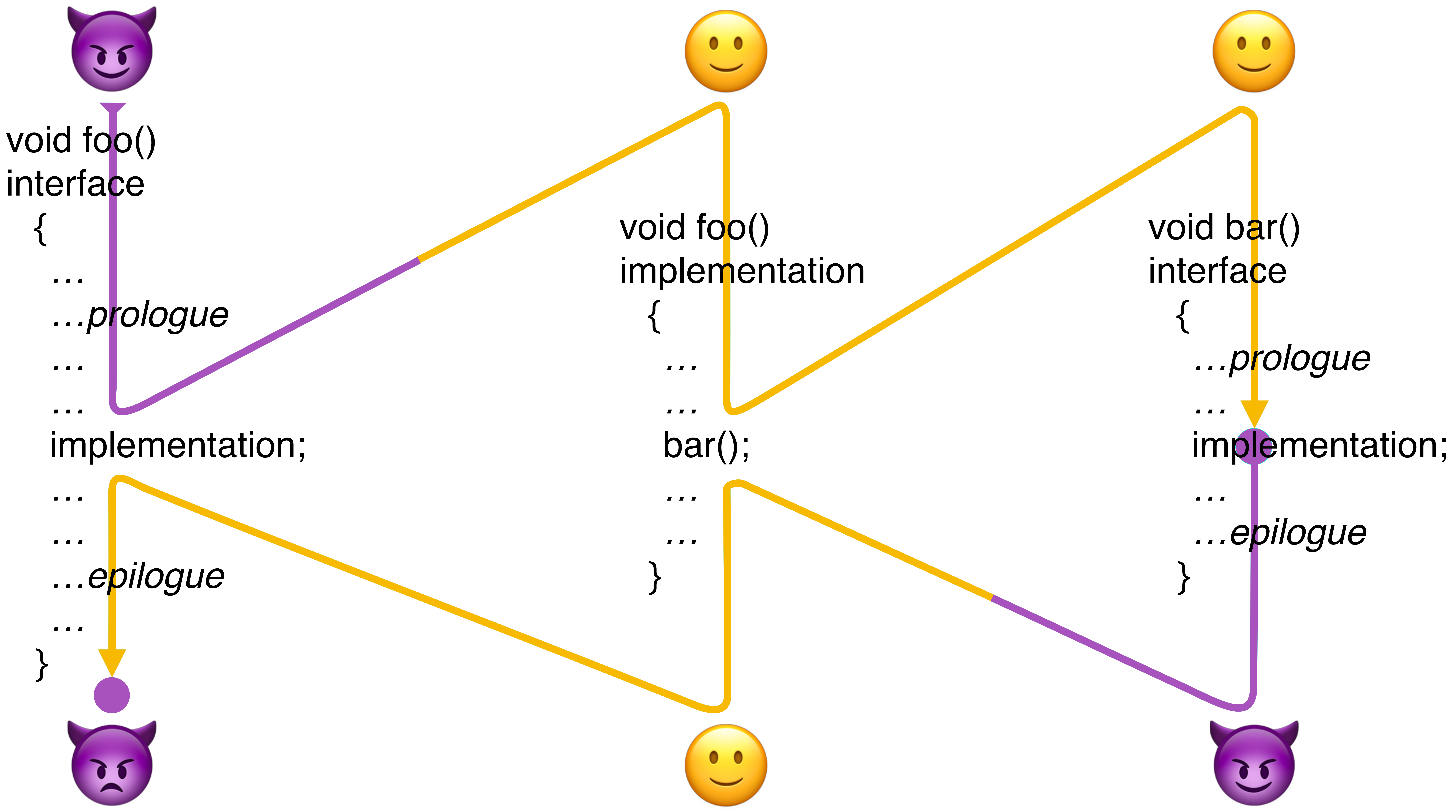
This sentence is **false:**
😈 has a winning strategy.



The code here is written in a fantasy C++, with extensions that make proofs fit into the code.



void foo()
interface
{
…
…prologue
…
…
implementation;
…
…
…epilogue
…
}
void foo()
implementation
{
…
…
bar();
…
…
}
void bar()
interface
{
…prologue
…
implementation;
…
…epilogue
}


void foo()
interface
{
…
…prologue
…
…
implementation;
…
…
…epilogue
…
}
void foo()
implementation
{
…
…
bar();
…
…
}
void bar()
interface
{
…prologue
…
implementation;
…
…epilogue
}


void foo()
interface
{
…
…prologue
…
…
implementation;
…
…
…epilogue
…
}
void foo()
implementation
{
…
…
bar();
…
…
}
void bar()
interface
{
…prologue
…
implementation;
…
…epilogue
}

```
const int factorial( const int& n )
interface
  {
   claim n >= 0;

   claim usable( n );

   implementation;

   claim usable( n );
   claim usable( result );
  }
```

```
const int factorial( const int& n )
interface
  {
    claim n >= 0;

    claim usable( n );

    implementation;

    claim usable( n );
    claim usable( result );
  }
```

claim statements are assertions that must hold *for local reasons.*

Yellow claims for reasons in this function;
purple claims for reasons in other functions.

👿🙁 If a **claim** statement fails, the current player loses.

```
const int factorial( const int& n )
interface
  {
   claim n >= 0;

   claim usable( n );

   implementation;

   claim usable( n );
   claim usable( result );
  }
```

An lvalue is **usable** if it may be used in the usual manner for its cv-qualified type.

Usable scalar lvalues
— have a stable value (if not volatile), and
— are modifiable (if not const).

Class types may have more complicated rules for usability.

```
const int factorial( const int& n )
interface
  {
   claim n >= 0;

   claim usable( n );

   implementation;

   claim usable( n );
   claim usable( result );
  }
```

If an operation is used in the procedure, its interface is part of the game.

We'll start the game with the interface of operator>=(const int&, const int&).

The current player announces the value of each **usable** lvalue.

The value of **a** is six.
And the value of **b** is zero.

```
const bool operator>=( const int& a,
                       const int& b )

interface
  {
   claim usable( a );
   claim usable( b );

   implementation;

   claim usable( a );
   claim usable( b );
   claim usable( result );
  }
```

If the object hasn't been changed, the player must repeat the previous value.

😈 The value of **a** is six.
And the value of **b** is zero.

😈 **a** is still six,
and **b** is still zero.
And the **result** is true.

👿🙁 Unexpectedly changing a value is penalized.

```
const bool operator>=( const int& a,
                       const int& b )
interface
  {
   claim usable( a );
   claim usable( b );

   implementation;

   claim usable( a );
   claim usable( b );
   claim usable( result );
  }
```

```
const int factorial( const int& n )
interface
   {
    claim n >= 0;

    claim usable( n );

    implementation;

    claim usable( n );
    claim usable( result );
   }
```

Lvalues asserted **usable** directly within the prologue provide the *direct input* to the function.

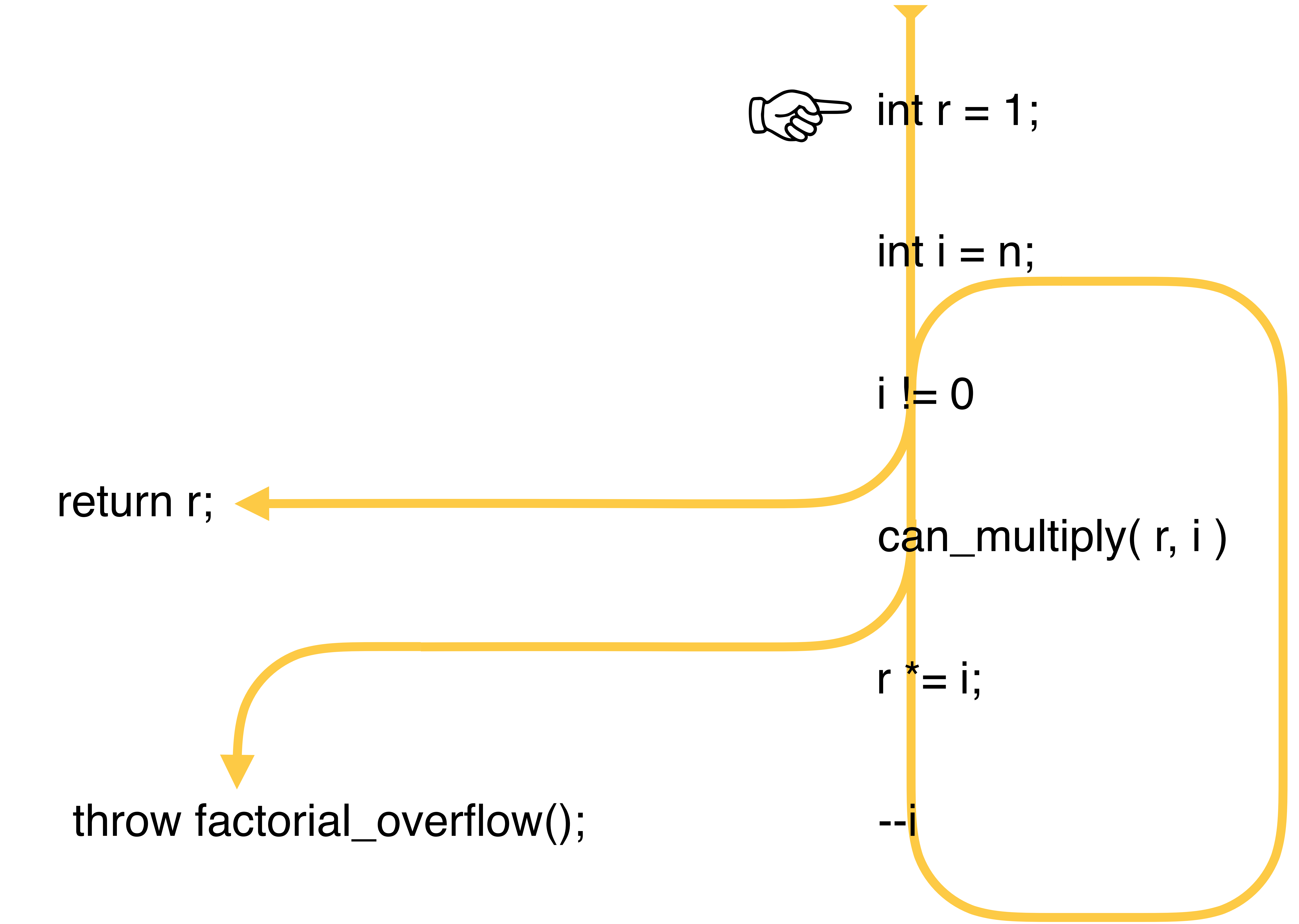
The result is **true**; the claim succeeds!

The value of **n** is six.

The epilogue likewise describes the *direct output.*

```
const int factorial( const int& n )
interface
  {
   claim n >= 0;

   claim usable( n );

   implementation;

   claim usable( n );
   claim usable( result );
  }
```

```
const int factorial( const int& n )
implementation
  {
   int r = 1;

   for ( int i = n;  i != 0;  --i )
     if ( can_multiply( r, i ) )
        r *= i;
      else
        throw factorial_overflow();

   return r;
  }
```


int r = 1;
int i = n;
i != 0
return r;
can_multiply(r, i)
r *= i;
throw factorial_overflow();
--i

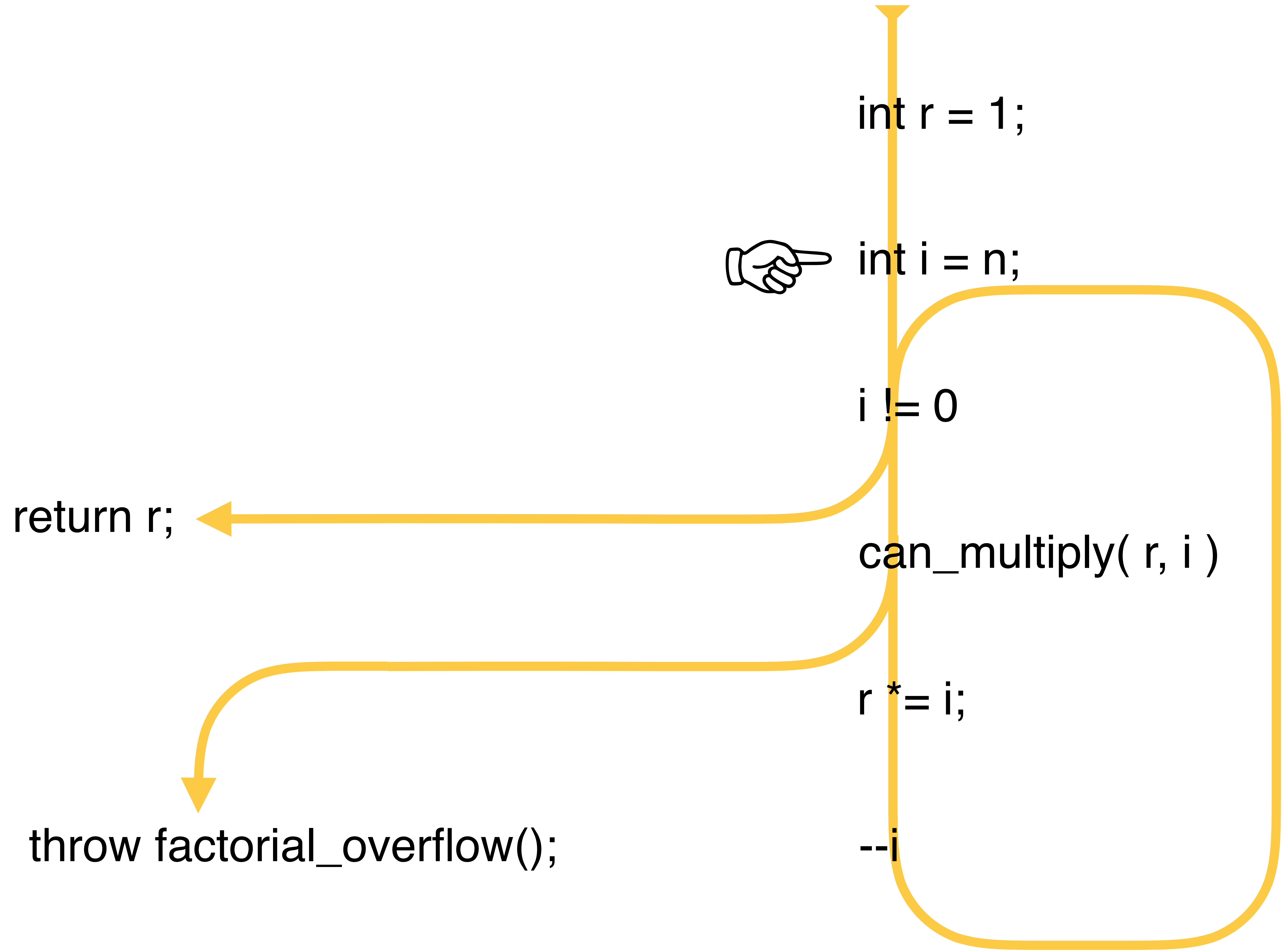
When substitutable is claimed,
lvalues must have identical values.

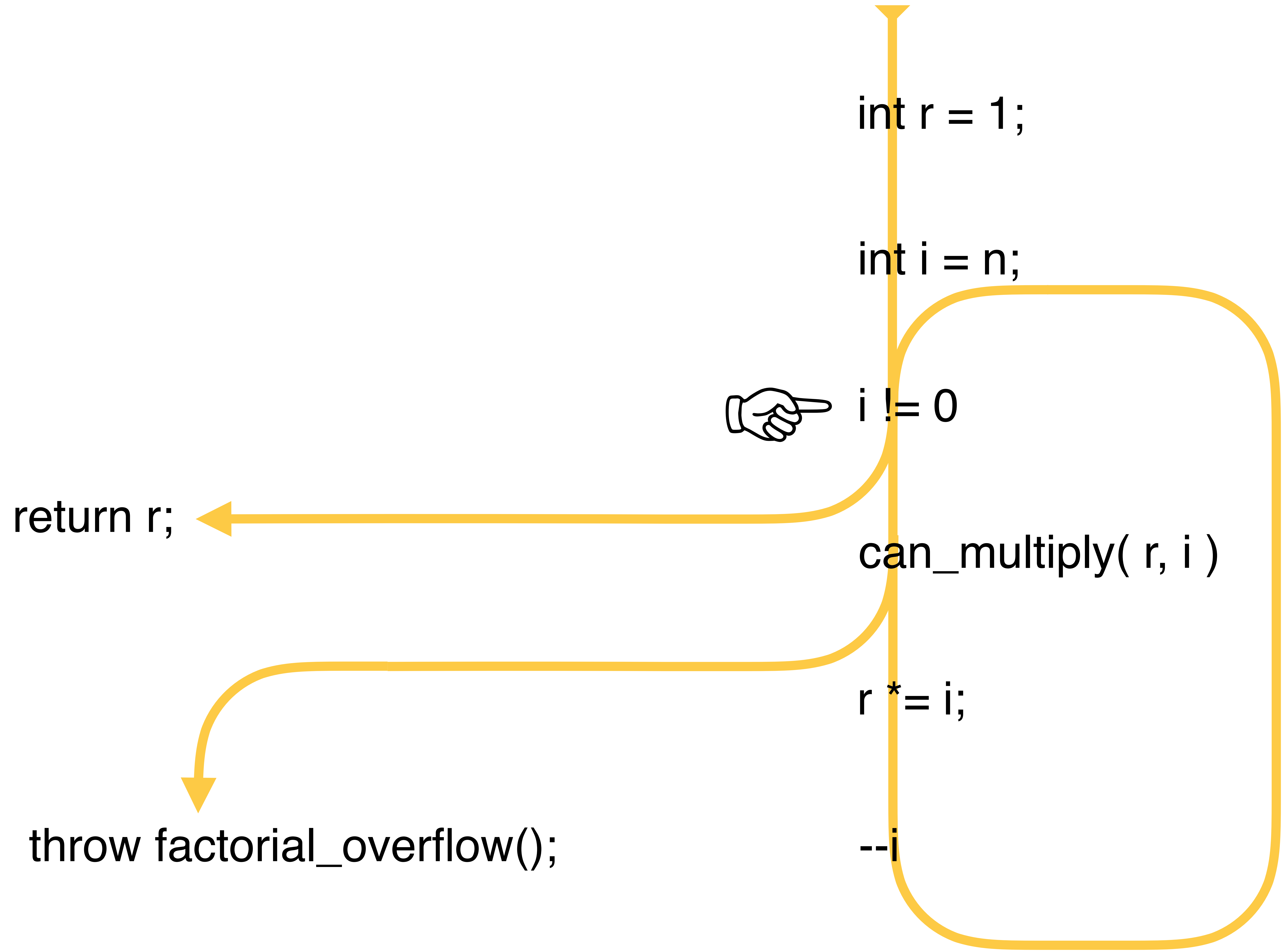
The value of **a** is one. 🙂

a and ***this** are both one.

The value of **a** is one, and
***this** is one. ***this** can be changed. 😈

```
int::int( const int& a )
interface
  {
   claim usable( a );

   implementation;

   claim substitutable( a, *this );

   claim usable( a );
   claim usable( *this );
  }
```


int r = 1;
int i = n;
i != 0
return r;
can_multiply(r, i)
r *= i;
throw factorial_overflow();
--i



int r = 1;
int i = n;
i != 0
return r;
can_multiply(r, i)
r *= i;
throw factorial_overflow();
--i

Inline functions without declared interfaces are played by the entering player.

Sometimes showing what a function does is simpler than describing it.
But this also makes the program brittle!

```
inline
const bool operator!=( const int& a,
                       const int& b )
  {
   return !( a == b );
  }
```

```
inline
const bool operator!( const bool& c )
  {
   return c ? false : true;
  }
```

Branch directions are also part of the direct input and output.

The value of **a** is six, and **b** is zero.

The **result** is false; swerve right!

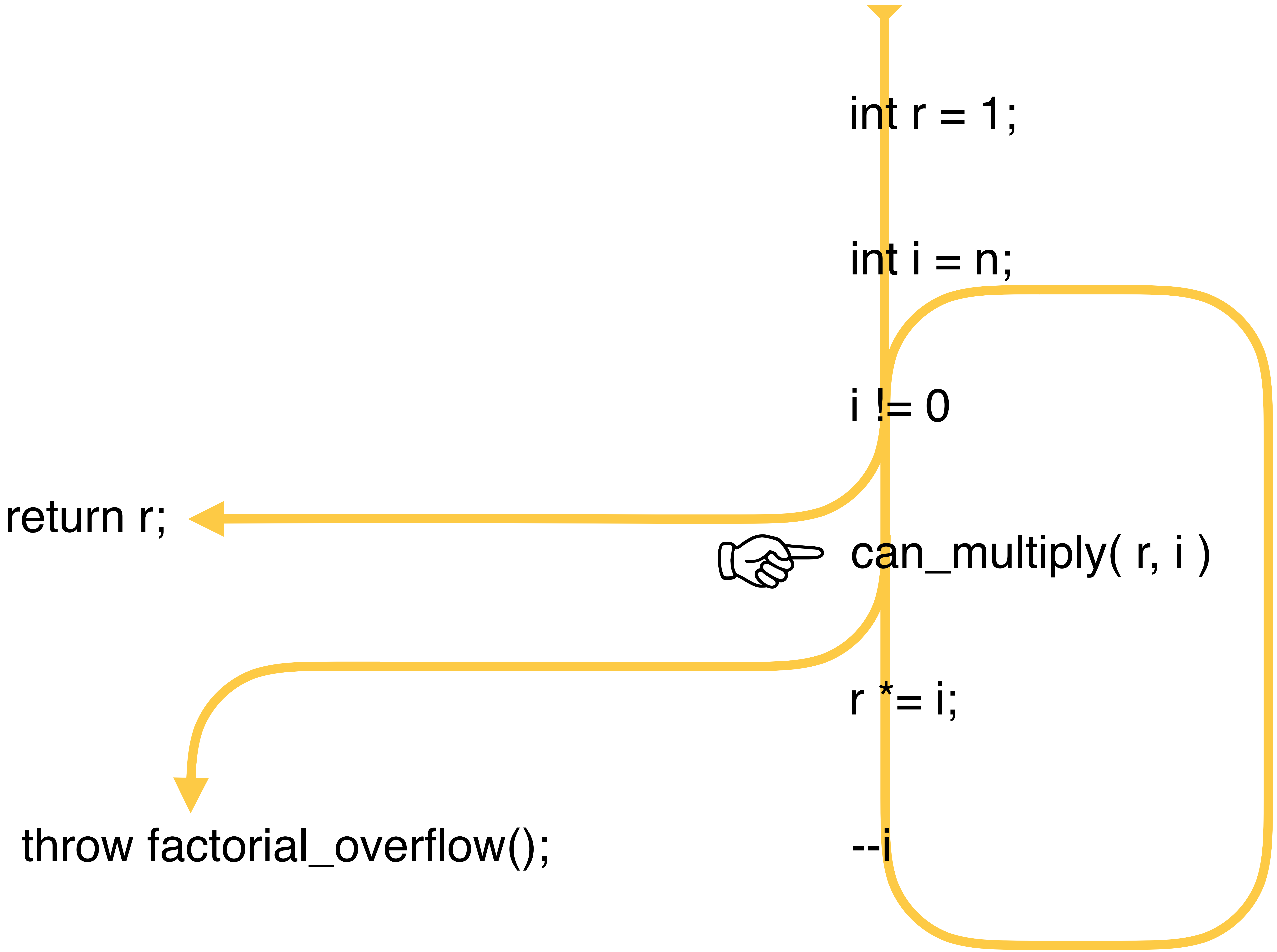
The value of **a** is still six,
b is still zero,
and the **result** is false.

```
const bool operator==( const int& a,
                       const int& b )

interface
  {
   claim usable( a );
   claim usable( b );

   implementation;

   if ( result )
      claim substitutable( a, b );

   claim usable( a );
   claim usable( b );
   claim usable( result );
  }
```

Inline functions without declared interfaces are played by the entering player.

Sometimes showing what a function does is simpler than describing it.
But this also makes the program brittle!

```
inline
const bool operator!=( const int& a,
                       const int& b )
  {
   return !( a == b );
  }

inline
const bool operator!( const bool& c )
  {
   return c ? false : true;
  }
```



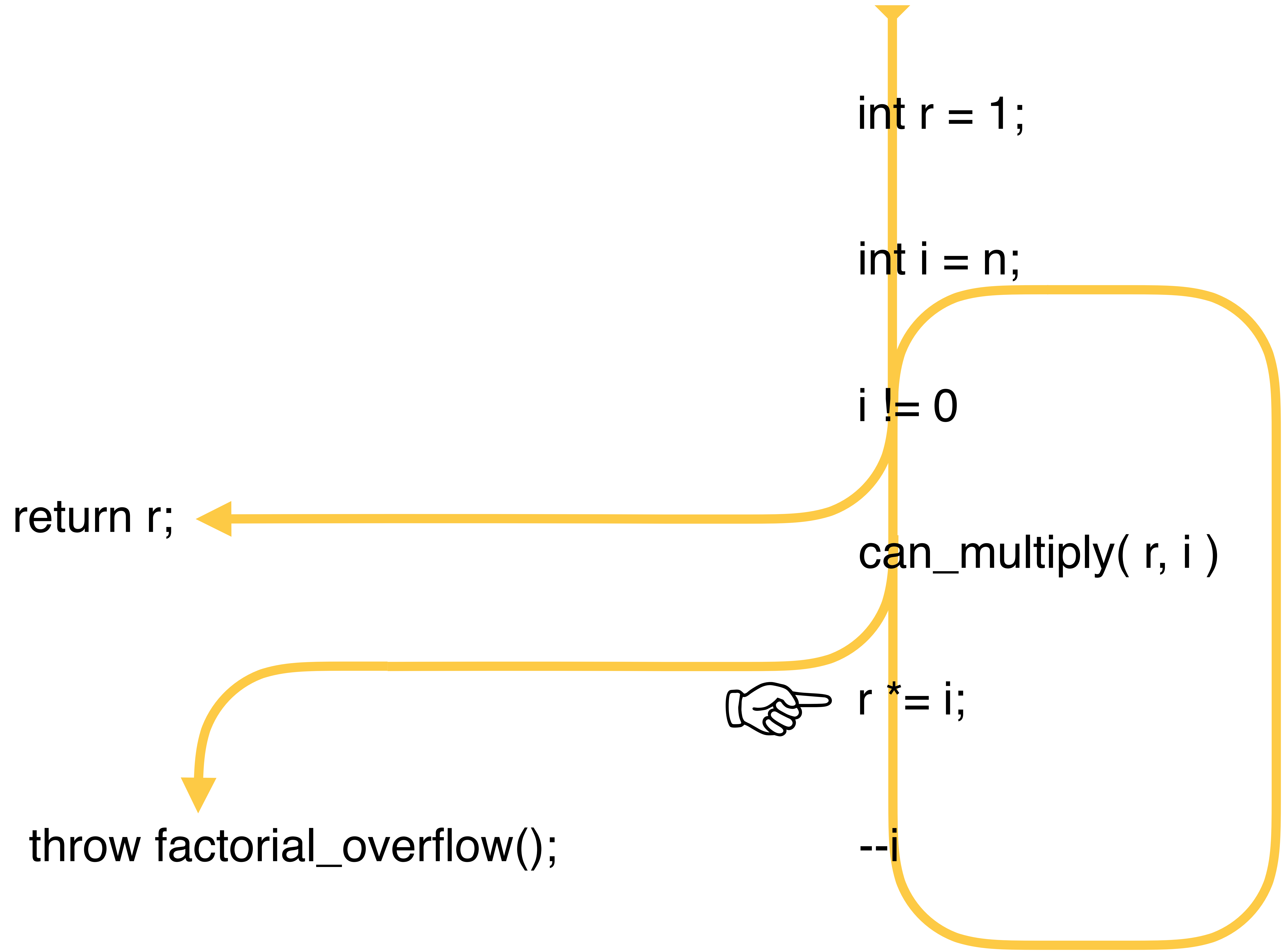
int r = 1;
int i = n;
i != 0
return r;
can_multiply(r, i)
r *= i;
--i
throw factorial_overflow();

can_multiply has a *basic interface:* usable input, usable output.

The value of **a** is one, and the value of **b** is six.

a is still one,
and **b** is still six.
And the **result** is true.

```
const bool can_multiply( const int& a,
                         const int& b )

interface
  {
    claim usable( a );
    claim usable( b );

    implementation;

    claim usable( a );
    claim usable( b );
    claim usable( result );
  }
```


int r = 1;
int i = n;
i != 0
return r;
can_multiply(r, i)
r *= i;
throw factorial_overflow();
--i

```
int& int::operator*=( const int m )
interface
  {
   claim can_multiply( *this, m );

   claim usable( m );
   claim usable( *this );

   implementation;

   claim aliased( result, *this );

   claim usable( m );
   claim usable( *this );
   claim usable( result );
  }
```

If a function's direct input is repeated, its direct output must also be repeated.

As before, the value of **a** is one, and the value of **b** is six. 🙂

😈 **a** is still one,
and **b** is still six.
Like last time, the **result** is true.

👿🙁 Announcing different direct output is penalized.

```
const bool can_multiply( const int& a,
                         const int& b )
interface
  {
   claim usable( a );
   claim usable( b );

   implementation;

   claim usable( a );
   claim usable( b );
   claim usable( result );
  }
```

Lvalues are aliased when they refer to the same object.

The **can_multiply** claim succeeds!

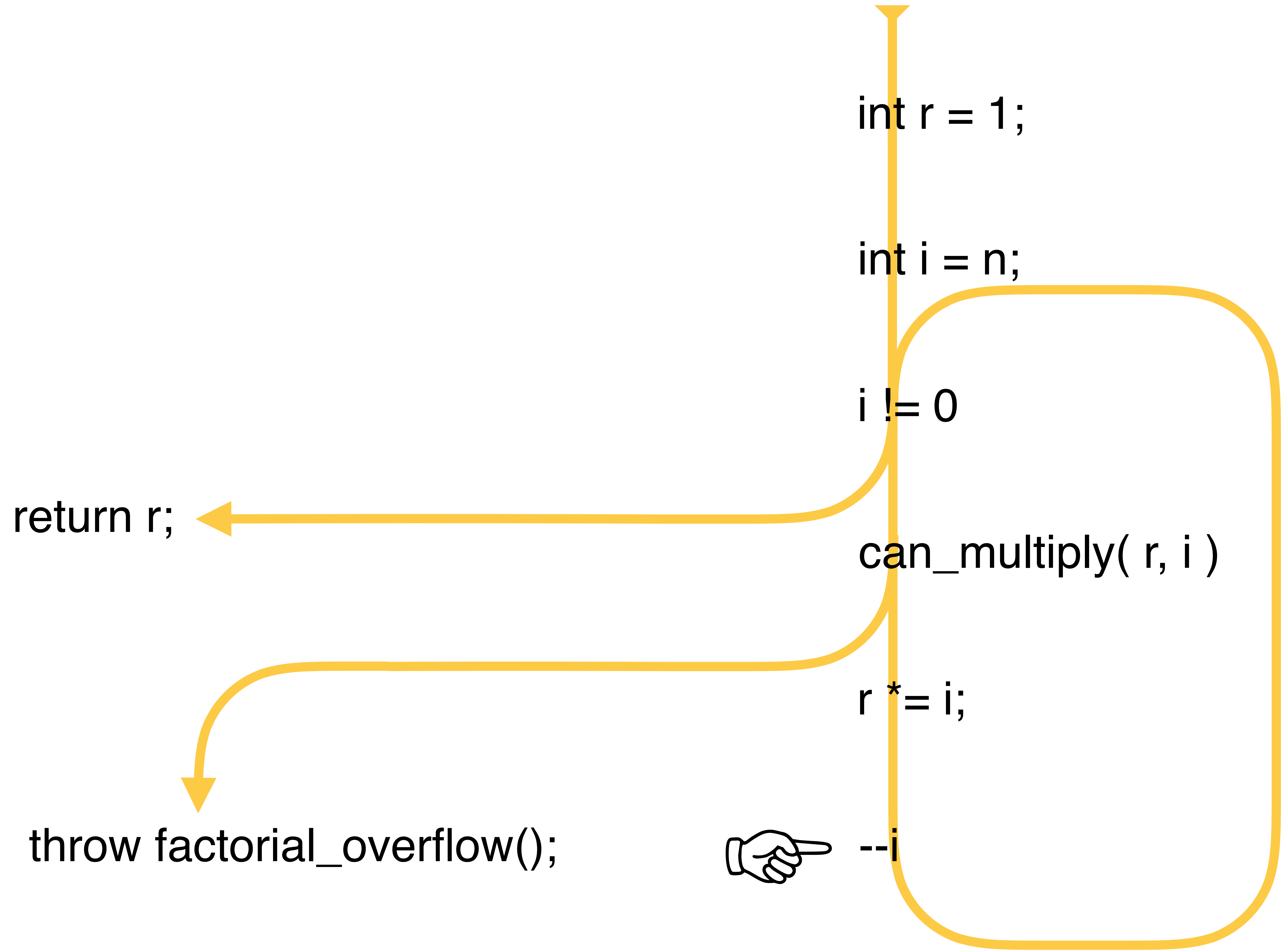
The value of **m** is six, and while ***this** is currently one, it can change. 🙂

result and ***this** are the same object.

😈 **m** is still six;
***this** is now six and can change;
the **result** is six and can change.

```
int& int::operator*=( const int m )
interface
  {
   claim can_multiply( *this, m );

   claim usable( m );
   claim usable( *this );

   implementation;

   claim aliased( result, *this );

   claim usable( m );
   claim usable( *this );
   claim usable( result );
  }
```

😈🙁 There is a penalty for *not* mentioning observable aliasing.


int r = 1;
int i = n;
i != 0
return r;
can_multiply(r, i)
r *= i;
throw factorial_overflow();
--i

```
const bool
can_decrement( const int& a )
interface
   {
    claim usable( a );
    implementation;
    claim usable( a );
    claim usable( result );
   }
```

Six.

Six.
True.

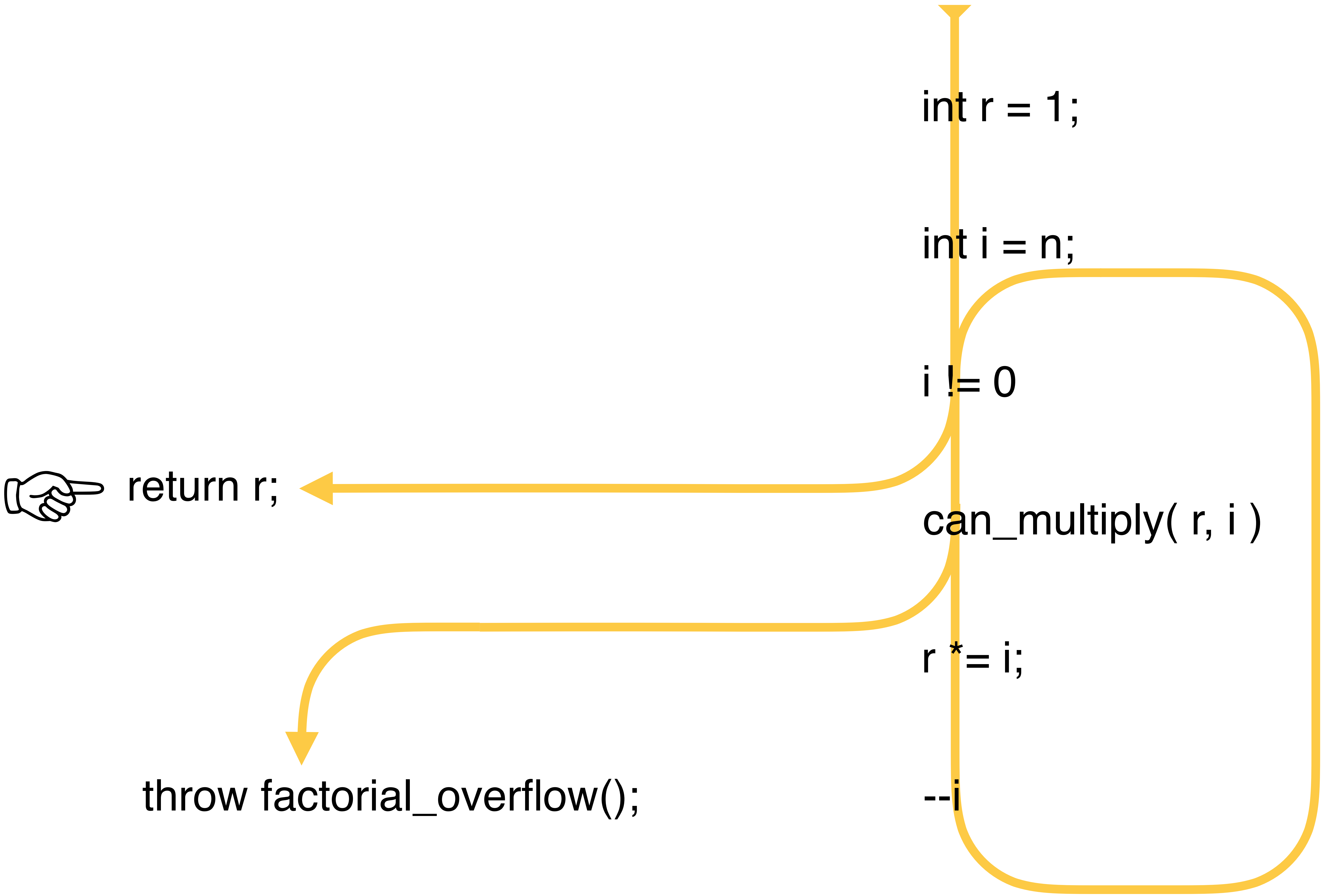
```
int& int::operator--()
interface
   {
    claim can_decrement( *this );
    claim usable( *this );
    implementation;
    claim can_increment( *this );
    claim aliased( *this, result );
    claim usable( *this );
    claim usable( result );
   }
```

Success!

Six; it changes.

Success!
Same object.

Both are now five;
they can change.


int r = 1;
int i = n;
i != 0
return r;
can_multiply(r, i)
r *= i;
throw factorial_overflow();
--i

```
const int factorial( const int& n )
interface
  {
   claim n >= 0;

   claim usable( n );

   implementation;

   claim usable( n );
   claim usable( result );
  }
```

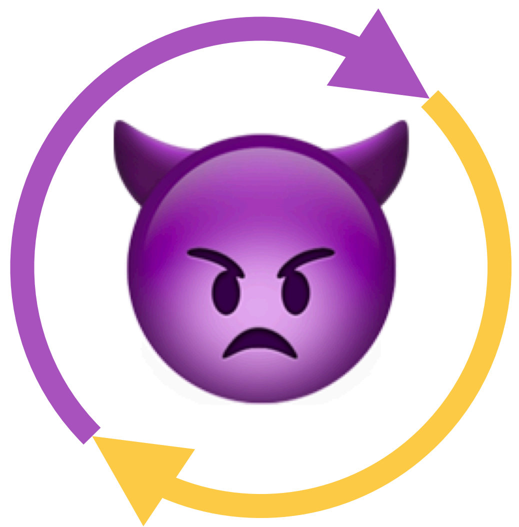
```
const int factorial( const int& n )
implementation
  {
   int r = 1;

   for ( int i = n;  i != 0;  --i )
      if ( can_multiply( r, i ) )
         r *= i;
      else
         throw factorial_overflow();

   return r;
  }
```

```
const int factorial( const int& n )
interface
  {
   claim n >= 0;

   claim usable( n );

   implementation;

   claim usable( n );
   claim usable( result );
  }
```

n is still six.
The **result** is seven hundred twenty.

```
const int factorial( const int& n )
interface
  {
   claim n >= 0;

   claim usable( n );

   implementation;

   claim usable( n );
   claim usable( result );
  }
```

Finally, 😈 can have rematches:
if 😈 repeats the direct input,
🙂 must repeat the direct output.



If this makes the game endless, 👿 loses.

n is still six.
The **result** is seven hundred twenty. 🙂

In the **game of truth**, 😈 announces the input, and 🙂 announces the output, broadly construed.

The game of truth has five penalty conditions:

- 😈🙁 Stuck in a loop
- 😈🙁 Assertion failure
- 😈🙁 Unexpected value change
- 😈🙁 Inconsistent function results
- 😈🙁 Unmentioned aliasing

🙂 **wins this game of truth**
if the first penalty goes to 👿.

😈 **wins this game of truth**
if the first penalty goes to 🙁.

🙂 wins this game of truth if the first penalty goes to 👿.

😈 wins this game of truth if the first penalty goes to 🙁.

🙂 **has a winning strategy** if the first penalty goes to 👿 **for all input values.**

😈 **has a winning strategy** if the first penalty goes to 🙁 **for some input values.**

🙂 wins this game of truth if the first penalty goes to 👿.

🙂 has a winning strategy if the first penalty goes to 👿 for all input values.

The procedure is true if 🙂 **has a winning strategy.**

😈 wins this game of truth if the first penalty goes to 🙁.

😈 has a winning strategy if the first penalty goes to 🙁 for some input values.

The procedure is false if 😈 **has a winning strategy.**

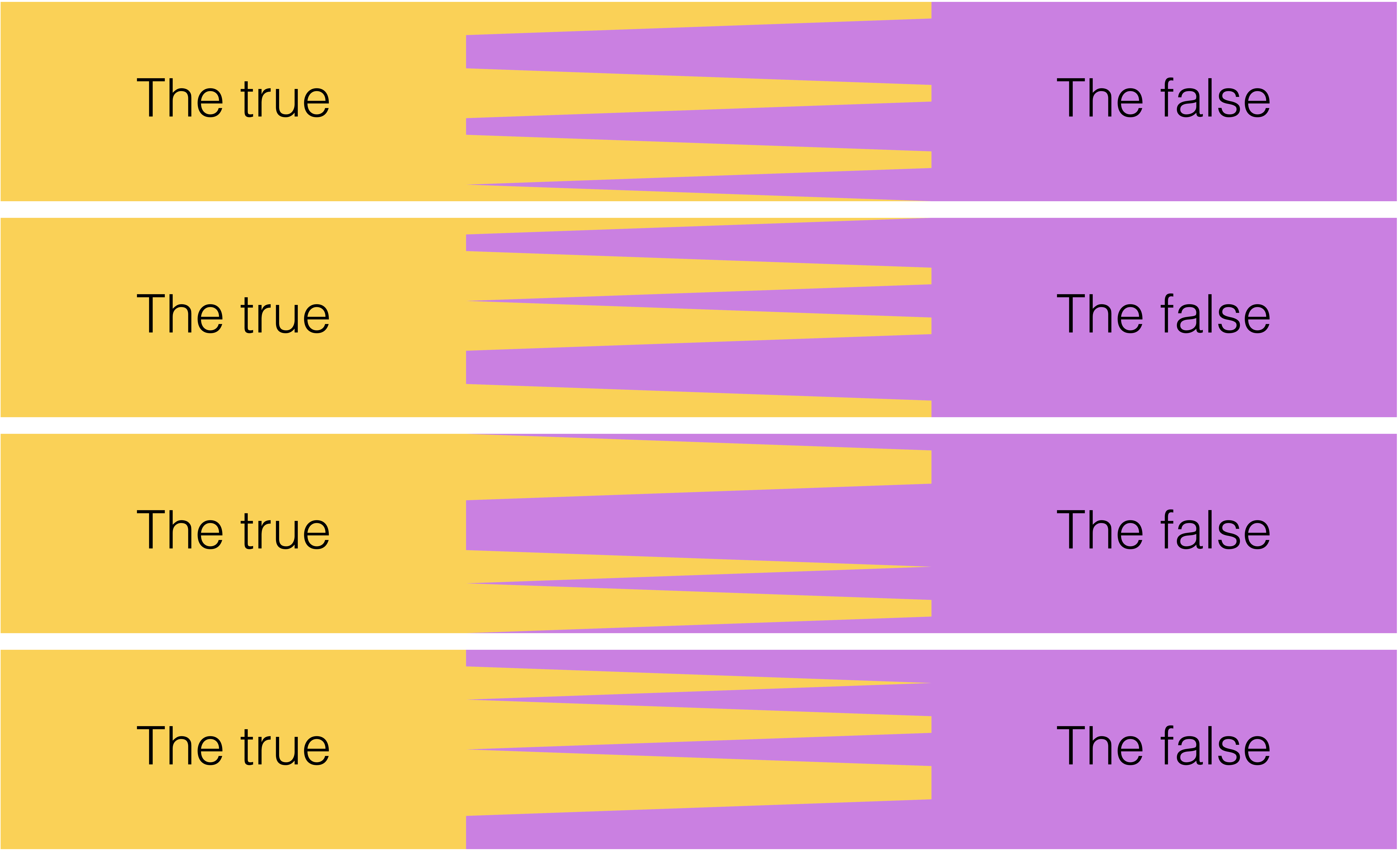
Q: Is there always a winning strategy for some player?
Or could a procedure be neither true nor false?

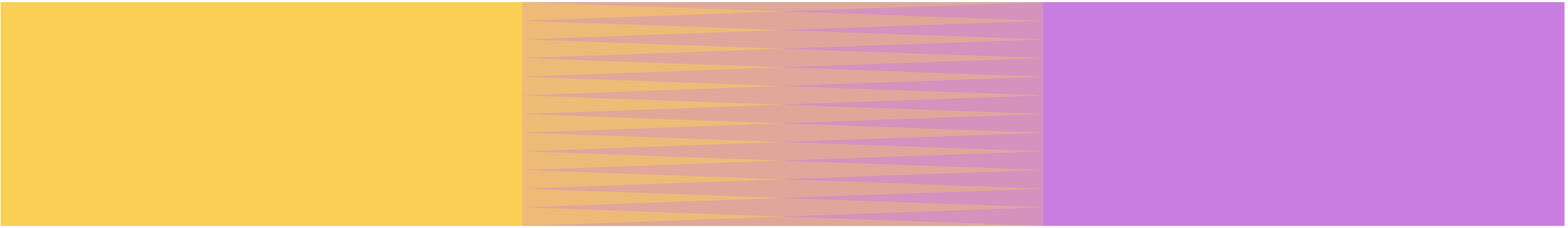
A: These games are topologically Borel. In a Borel game, if one player does not have a winning strategy, the other player does.

(“Borel determinacy,” Donald A. Martin, 1975)

✅ Euclidean geometry
✅ Algebraically closed fields (of any characteristic)
✅ Dense linear orderings (with or without endpoints)

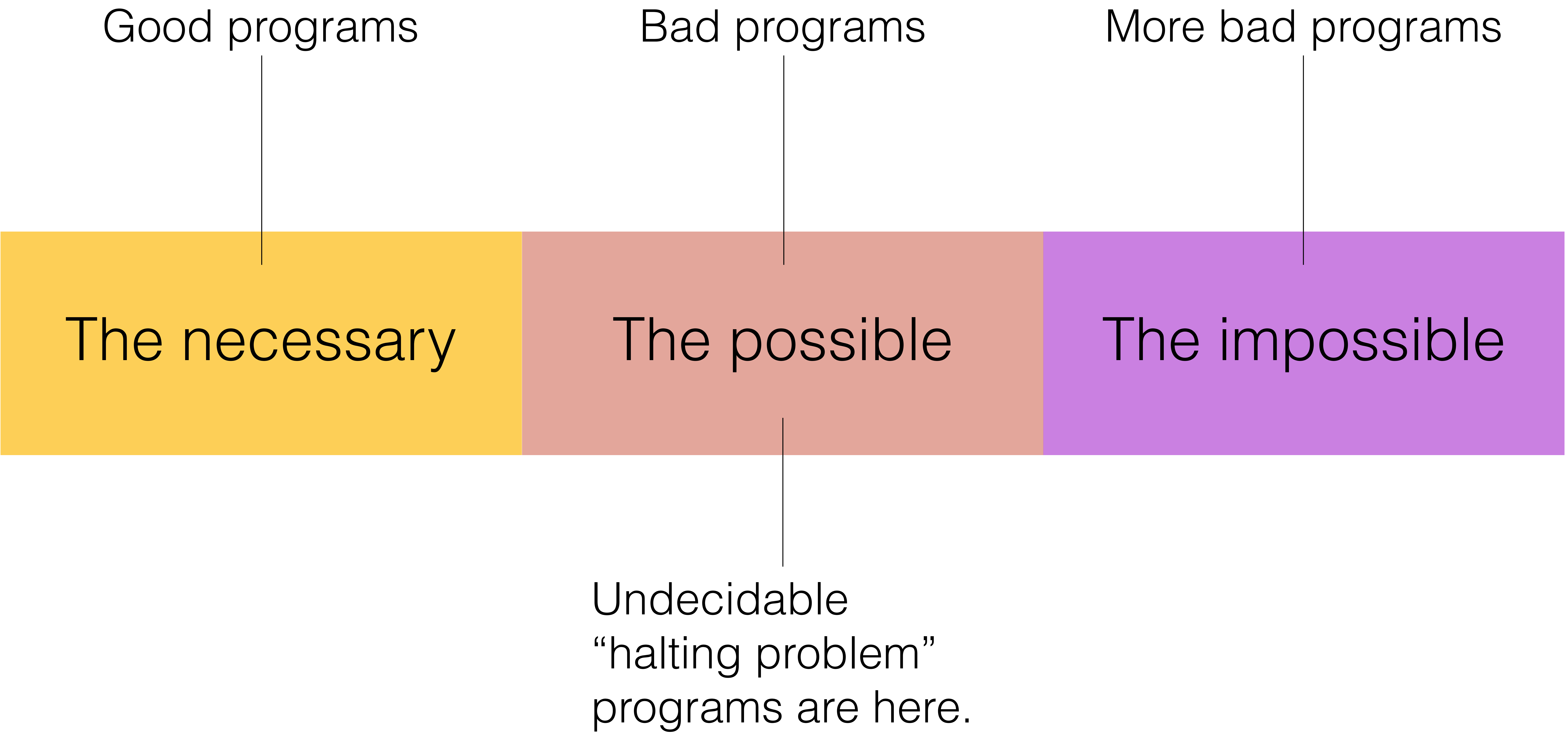
The true	The false

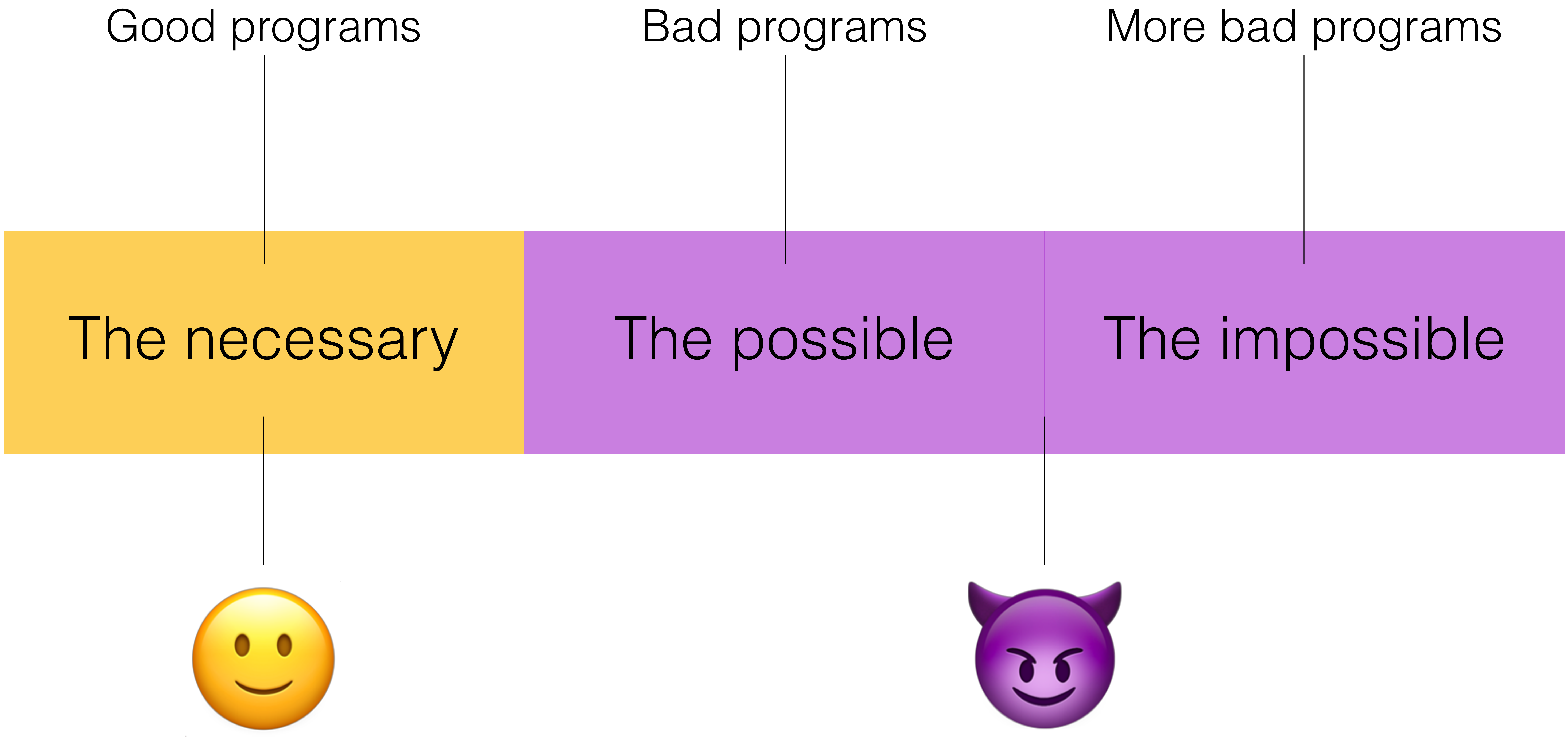

The true
The false
The true
The false
The true
The false
The true
The false



The necessary	The possible	The impossible


The necessary
The possible
The impossible
Undecidable
“halting problem”
programs are here.


Good programs
Bad programs
More bad programs
The necessary
The possible
The impossible
Undecidable
“halting problem”
programs are here.


Good programs
Bad programs
More bad programs
The necessary
The possible
The impossible

Q: Is there some advantage we can give to 😈 so that 🙂 wins only if the procedure is necessarily true?

A: We can put 😈 in charge of the computer!
That's the principle behind the **game of necessity.**

Instead of choosing values,
😈 *names* the usable values.

😈 The value of **a** is Sue.
And the value of **b** is Zachary.

```
const bool operator>=( const int& a,
                       const int& b )
interface
  {
   claim usable( a );
   claim usable( b );

   implementation;

   claim usable( a );
   claim usable( b );
   claim usable( result );
  }
```

If the object hasn't been changed, 😈 must repeat the previous name.

😈 The value of **a** is Sue.
And the value of **b** is Zachary.

😈 **a** is still Sue,
and **b** is still Zachary.
And the **result** is Bob. Bob the boolean.

```
const bool operator>=( const int& a,
                       const int& b )

interface
  {
   claim usable( a );
   claim usable( b );

   implementation;

   claim usable( a );
   claim usable( b );
   claim usable( result );
  }
```

At branches and claims, 😈 tells us which way to go.

```
const int factorial( const int& n )
interface
  {
   claim n >= 0;

   claim usable( n );

   implementation;

   claim usable( n );
   claim usable( result );
  }
```

😈 Bob is a left-turning boolean; the claim succeeds!

😈 must be consistent: once a boolean turns one way, it must always turn that way.

When claiming substitutability, 😈 explains that both names refer to the same value.

The value of **a** is Sam, and the value of **b** is Fred. 🙂

Swerve left!

Fred is Sam's middle name.

Sammy-Freddy, his parents used to call him.

😈 True story!

```
const bool operator==( const int& a,
                       const int& b )

interface
  {
   claim usable( a );
   claim usable( b );

   implementation;

   if ( result )
      claim substitutable( a, b );

   claim usable( a );
   claim usable( b );
   claim usable( result );
  }
```

Instead of announcing values,
🙂 repeats names used by 😈.

```
claim usable( f );
```

That's good old Charlie. 🙂

If the value wasn't named in
some previous claim, 🙁 loses.

```
claim usable( v );
```

??? 🤔

At branches and boolean claims, 🙂 asks 😈 which way to go.

```
if ( can_multiply( r, i ) )
```

Which way does Betty turn? 🙂

😈 Betty turns left at branches.

If 😈 hasn't already chosen a left turn, a boolean claim may not go well for 🙁.

```
claim decrementable( a );
```

Which way does Eddie turn? 😟

😈 Right! The claim fails!

When claiming substitutability, 🙂 reminds 😈 that both names refer to the same value.

claim substitutable(x, y);

And here's Forn, who you say is called Orald by the northern men. 🙄

If the names differ, and 😈 didn't already claim substitutability, 🙁 loses.

claim substitutable(p, q);

Could Bacon be Shakespeare? 🤯

In the **game of truth**, 😈 announces the input, and 🙂 announces the output, broadly construed.

In the **game of necessity**, 😈 tells a story, and 🙂 tells how the procedure executes within the story.

The game of necessity has seven penalty conditions:

- 😈🙁 Stuck in a loop
- 😈🙁 Assertion failure
- 😈🙁 Unexpected name change
- 😈🙁 Inconsistent result names
- 😈🙁 Unmentioned aliasing
- 😈 Inconsistent branches
- 🙁 Novel atomic claim

🙂 has a winning strategy for this **game of necessity** if the procedure is **true for all possible computers.**

😈 has a winning strategy for this **game of necessity** if the procedure is **false for some possible computer.**

(Forcing, Paul Cohen, 1963)

```
int& int::operator--()
interface
  {
   claim can_decrement( *this );
```

Which way?

```
   claim usable( *this );

   implementation;

   claim can_increment( *this );
   claim aliased( *this, result );

   claim usable( *this );
   claim usable( result );
  }
```

Right turn!
You lose.

```
const bool
can_decrement( const int& a )
interface
  {
   claim usable( a );

   implementation;

   claim usable( a );
   claim usable( result );
  }
```

Sue.

Sue.
Eddie.

Q: Is there some advantage we can give to 🙂 that's stronger than putting 😈 in charge of the computer?

A: We can team up with 🙂 to write the procedure! That's the principle behind the **game of proof.**

```
const int factorial( const int& n )
implementation
  {
   int r = 1;

   claim countdown_theorem( n, 0 );   🙂

   for ( int i = n;  i != 0;  --i )
     if ( can_multiply( r, i ) )
        r *= i;
      else
        throw factorial_overflow();

   return r;
  }
```

In this game, 🙂 can insert **claim** statements into the function implementation as the game is being played.

The new claims can include calls to **claimable** functions implemented elsewhere.

Such functions don’t affect execution, but just explain logical connections.

(Logicians call them “theorems.”)

```
claimable
countdown_throrem( const int& high,
                   const int& low )
interface
  {
   claim high >= low;

   claim implementation;

   for ( int c = high;  c != low;  --c )
     {}
  }
```

How do you count down from Sue to Zachary?

To sum up: Sue >= Zachary is Bob. Which way does Bob turn?

As I said before, Bob turns left.

Sue, Frank, Faye, Ted, Terry, Ollie, and the loop ends with Zachary.

```
claimable
countdown_throrem( const int& high,
                   const int& low )
interface
  {
   claim high >= low;

   claim implementation;

   for ( int c = high;  c != low;  --c )
      {}
  }
```

In the **game of truth**, 😈 announces the input, and 🙂 announces the output, broadly construed.

In the **game of necessity**, 😈 tells a story, and 🙂 tells how the procedure executes within the story.

In the **game of proof**, 😈 tells a story while 🙂 asks questions, forcing 😈 to expand on the story.

🙂 has a winning strategy for this **game of proof** if the procedure can be made necessary by **adding claims to the implementation.**

(Compactness)

😈 has a winning strategy for this **game of proof** if the procedure is false for some possible computer **that obeys the claimable rules.**

(Forcing, filtered colimits, finite injury)

Cf. Completeness, Kurt Gödel, 1929

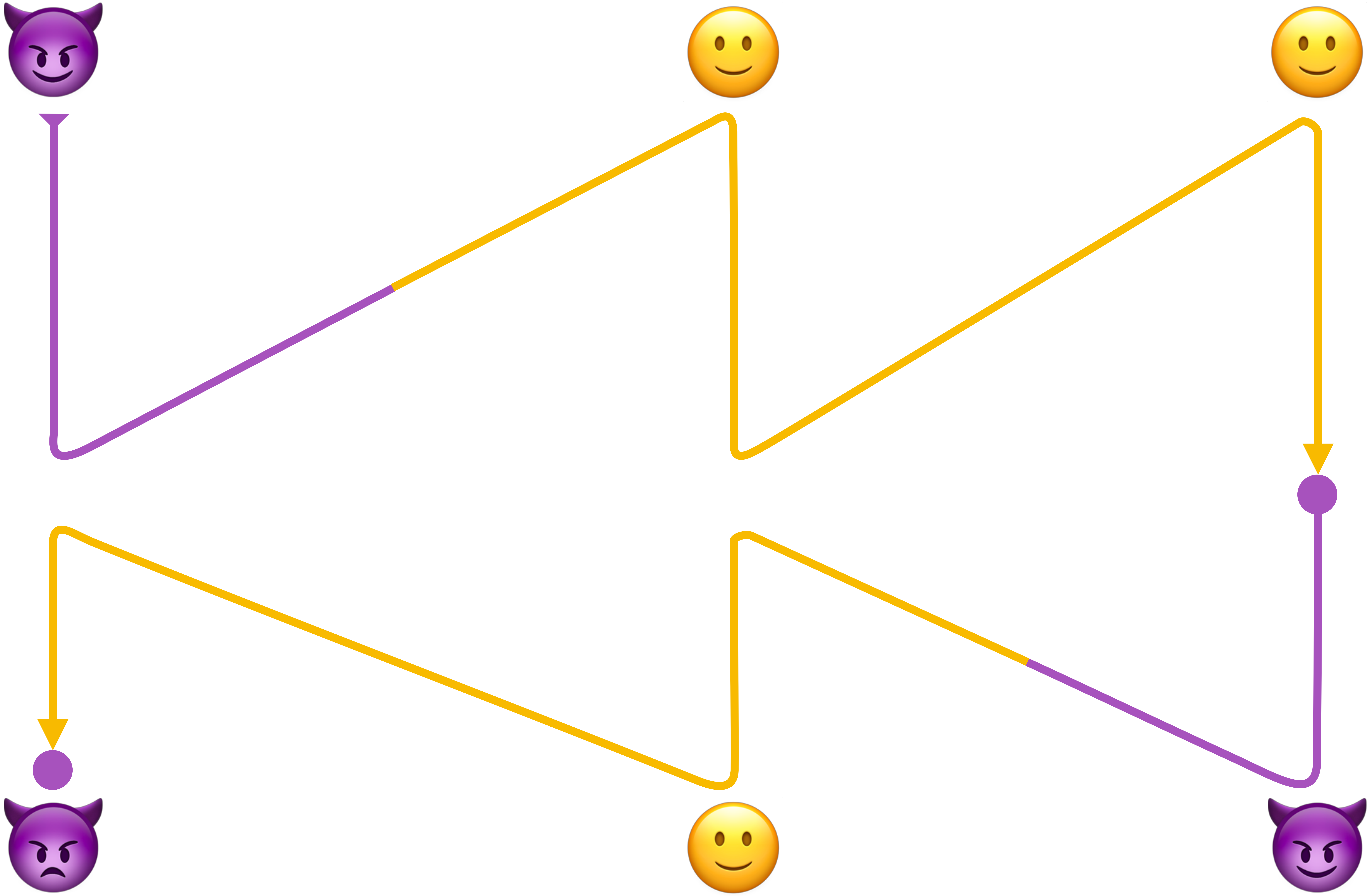
```
const int factorial( const int& n )
interface
  {
   claim n >= 0;

   claim usable( n );

   implementation;

   claim usable( n );
   claim usable( result );
  }
```

The trouble came from not saying what we meant at this point.

```
const int factorial( const int& n )
interface
  {
   for ( int i = n;  i != 0;  --i )
     {}

   claim usable( n );

   implementation;

   claim usable( n );
   claim usable( result );
  }
```

The trouble came from not saying what we meant at this point.

If the interface had expressed the precondition the function really used, there would have been no need to call a theorem.



In the big picture, there are no demons.

There are only other players, trying to win their own games.

Questions?