

# Forward Chaining in HALO

## An Implementation Strategy for History-based Logic Pointcuts

---

@International Conference On Dynamic Languages 2007,  
Lugano, August 27th, 2007

Charlotte Herzeel, Kris Gybels, Pascal Costanza, Coen De Roover and Theo D'hondt  
Programming Technology Laboratory  
Vrije Universiteit Brussel



# Overview

- Introduction
  - Logic Meta Programming & Aspect-Oriented Programming
  - Running example - going shopping
- **History-based Aspects using LOGic:**
  - Aspect-Oriented Programming for Lisp
  - meta model = the program execution
  - hybrid language (temporal logic programming/CLOS)
- **Implementing HALO**
  - **HALO weaver Architecture**
  - **Query Engine based on RETE**
  - **Reducing Memory Overhead**

# Scattering & tangling

# Scattering & tangling

Tyranny of the dominant decomposition

# Scattering & tangling

Tyranny of the dominant decomposition



tangling & scattering

- maintainability
- reusability
- readability

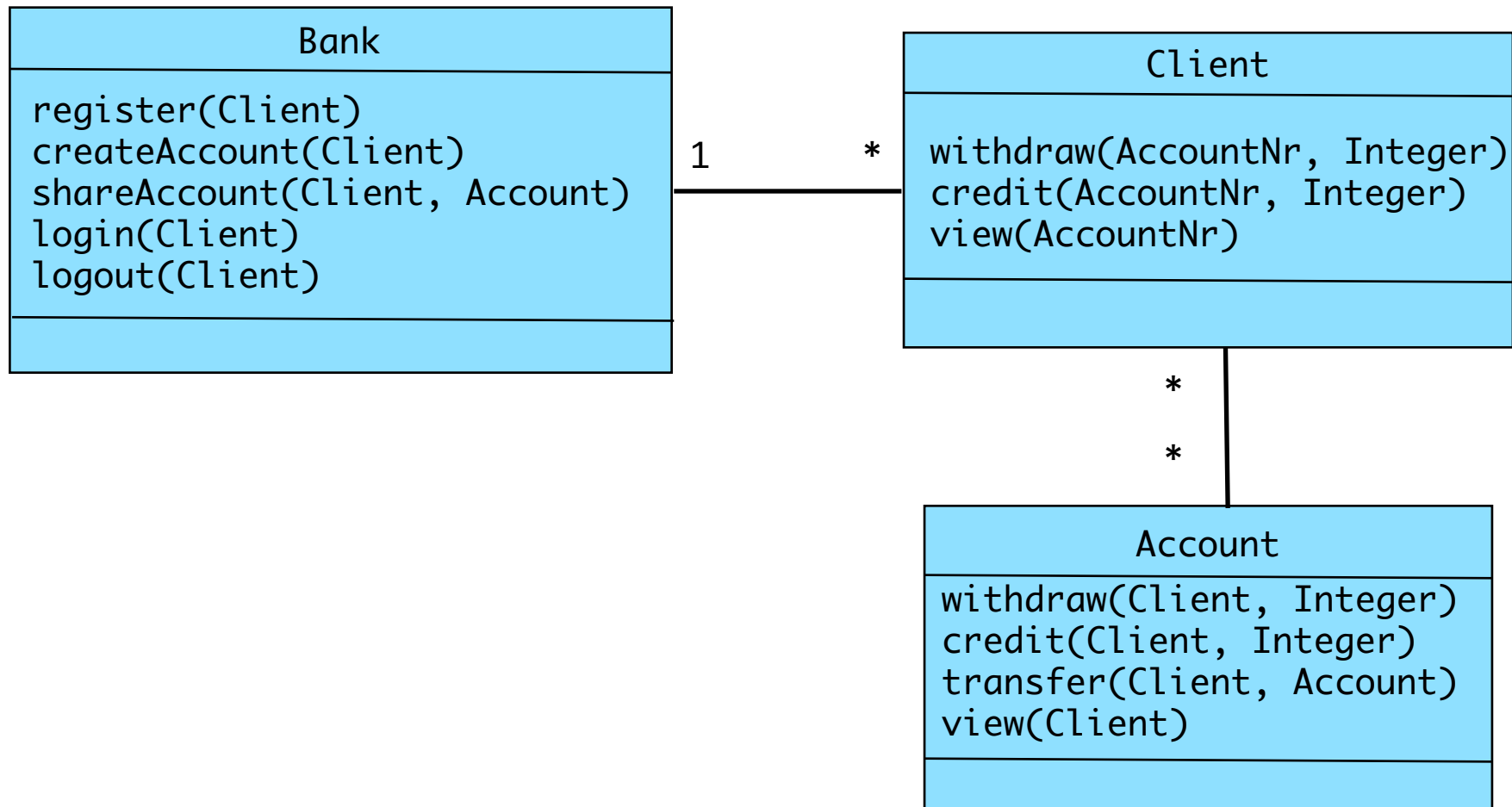
# Scattering & tangling

Tyranny of the dominant decomposition



tangling & scattering

- maintainability
- reusability
- readability



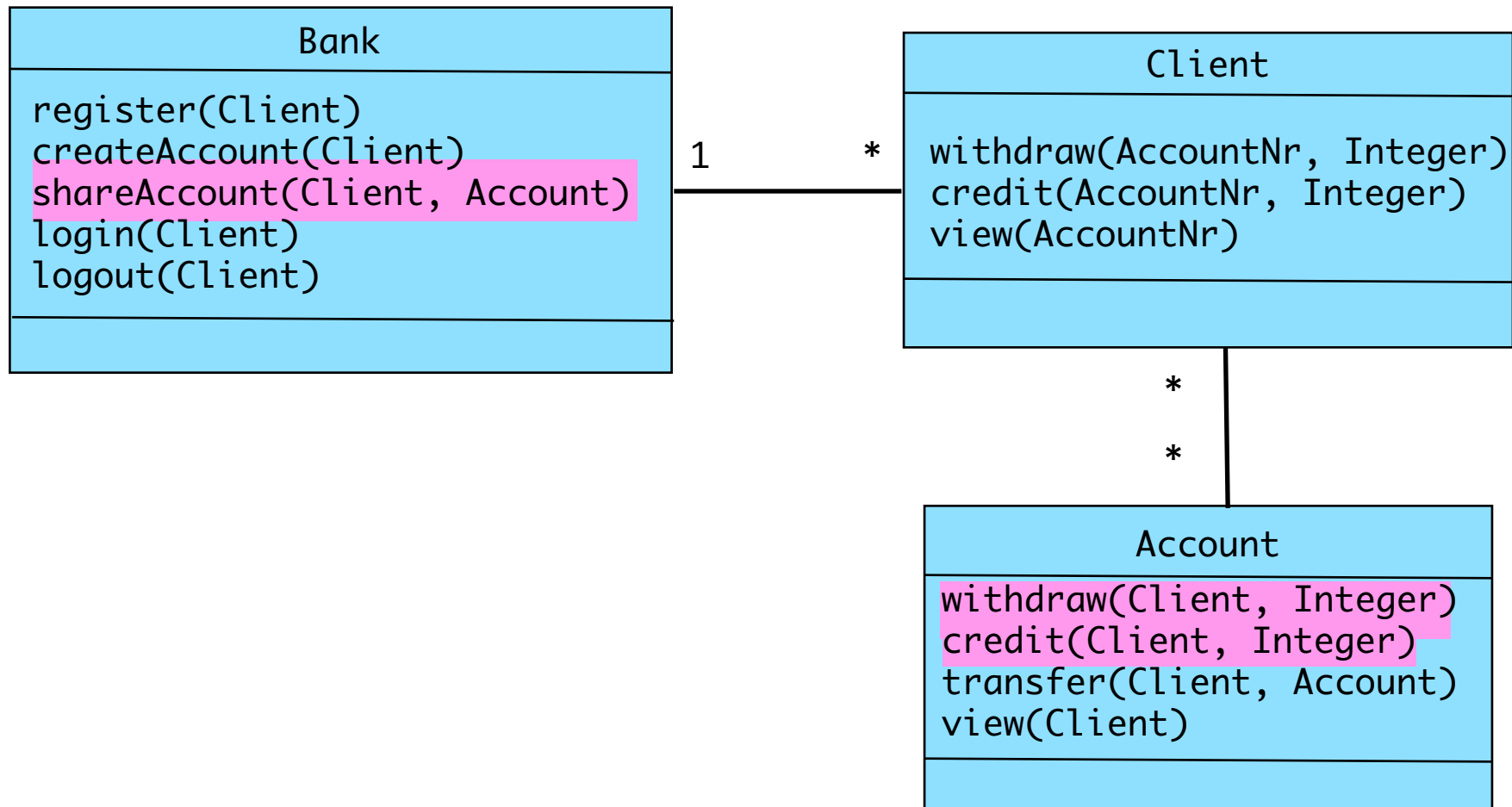
# Scattering & tangling

Tyranny of the dominant decomposition



tangling & scattering

- maintainability
- reusability
- readability



# Scattering & tangling

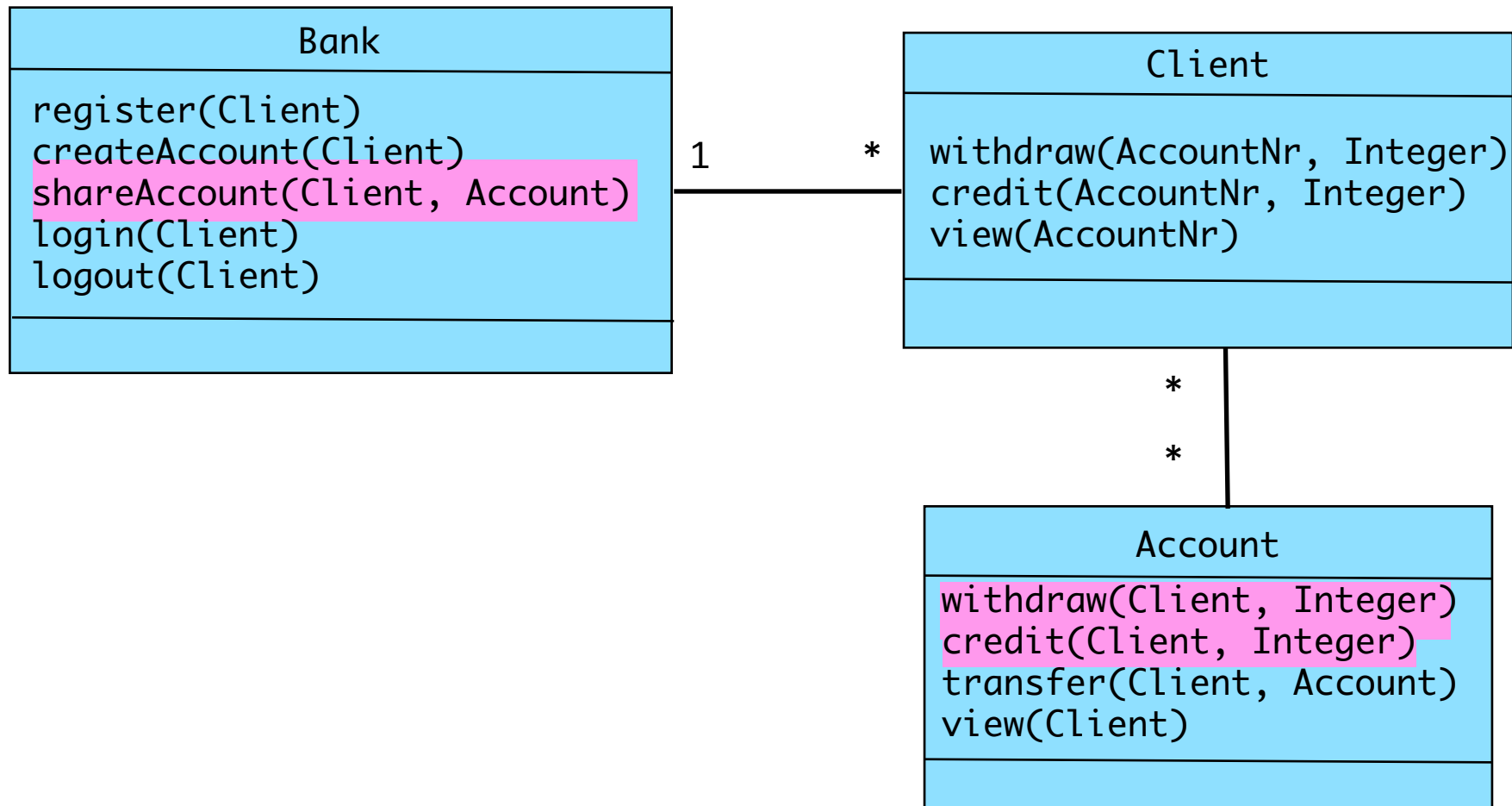
log sensitive operations = crosscutting concern

Tyranny of the dominant decomposition



tangling & scattering

- maintainability
- reusability
- readability





# Scattering & tangling

log sensitive operations = crosscutting concern

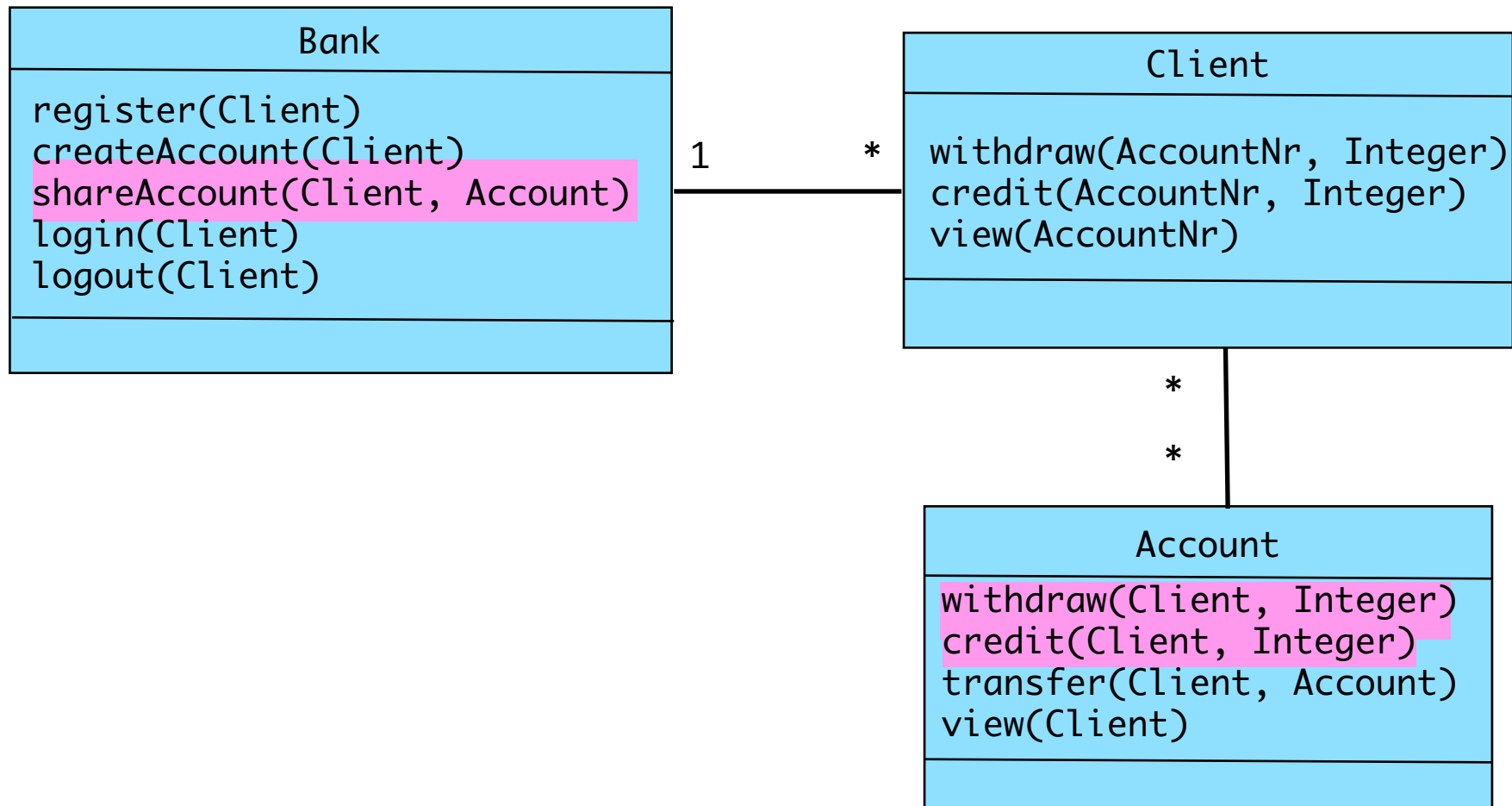
Tyranny of the dominant decomposition



tangling & scattering

- maintainability
- reusability
- readability

CLOS: before, after, around



# Scattering & tangling

log sensitive operations = crosscutting concern

Tyranny of the dominant decomposition



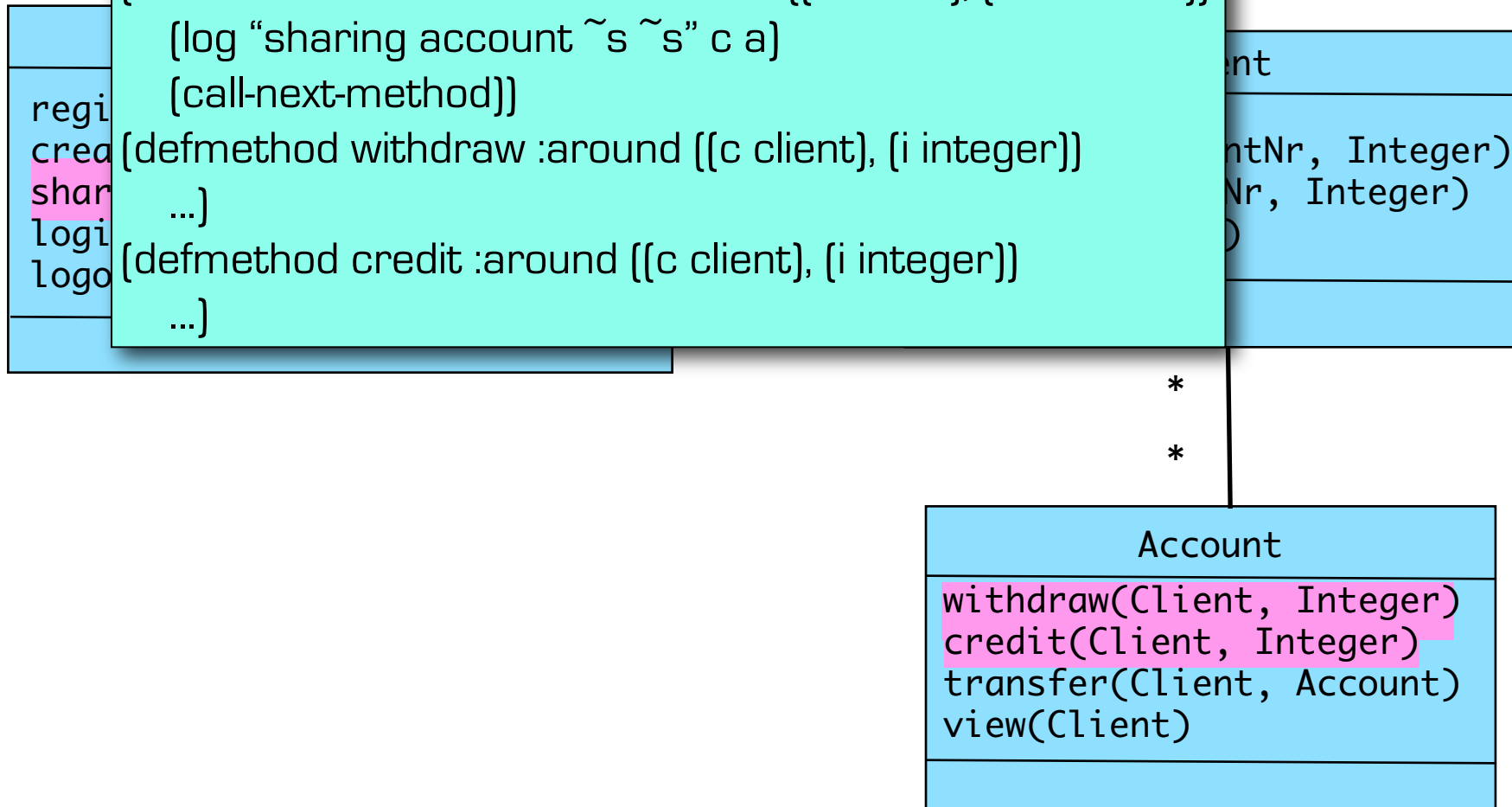
tangling & scattering

- maintainability
- reusability
- readability

CLOS: before, after, around



```
(defmethod shareAccount :around [(c client), (a account)]  
  (log "sharing account ~s ~s" c a)  
  (call-next-method))  
(defmethod withdraw :around [(c client), (i integer)]  
  ...)  
(defmethod credit :around [(c client), (i integer)]  
  ...)
```



# Scattering & tangling

log sensitive operations = crosscutting concern

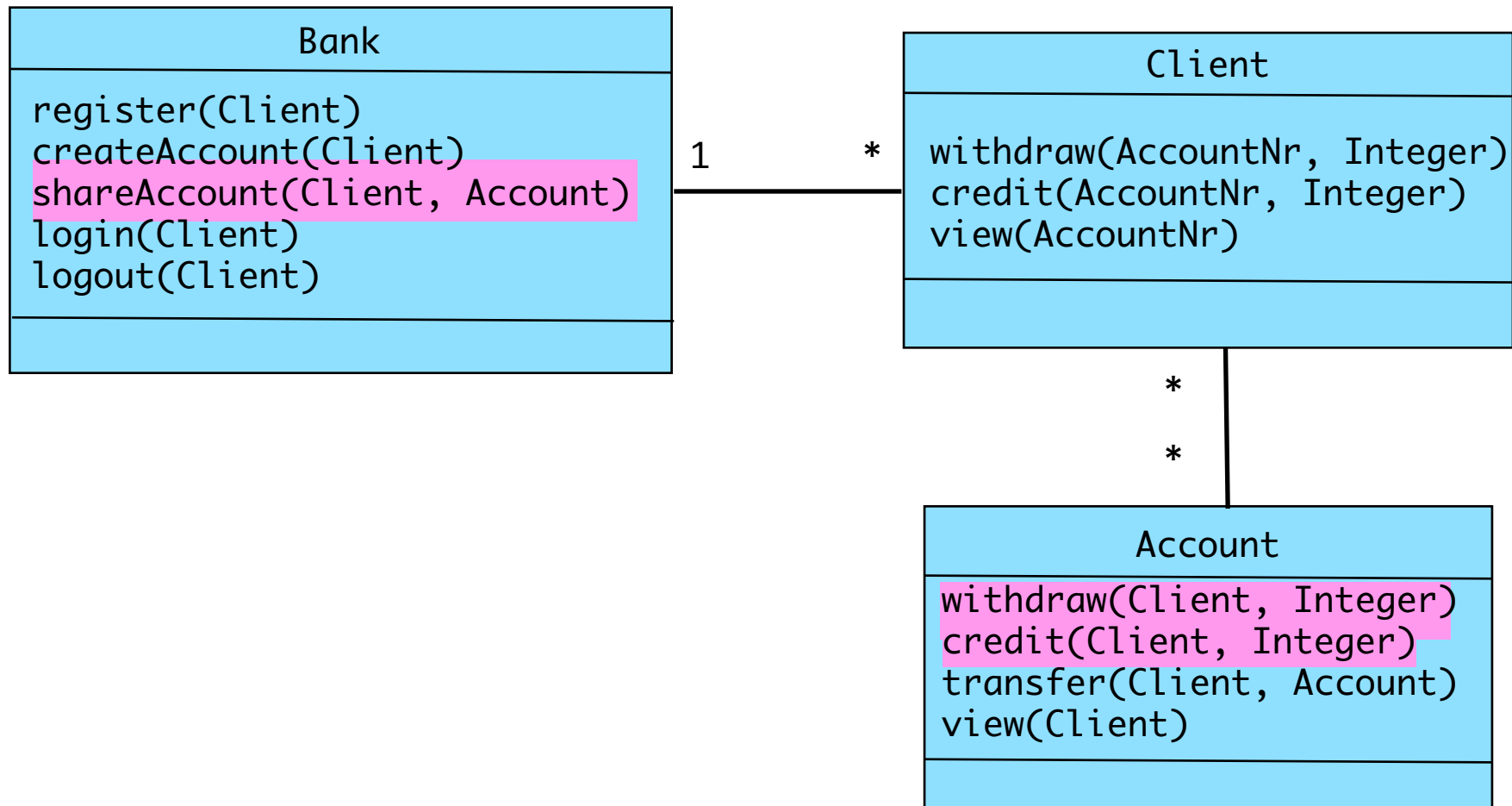
Tyranny of the dominant decomposition



tangling & scattering

- maintainability
- reusability
- readability

CLOS: before, after, around



# Scattering & tangling

log sensitive operations = crosscutting concern

Tyranny of the dominant decomposition



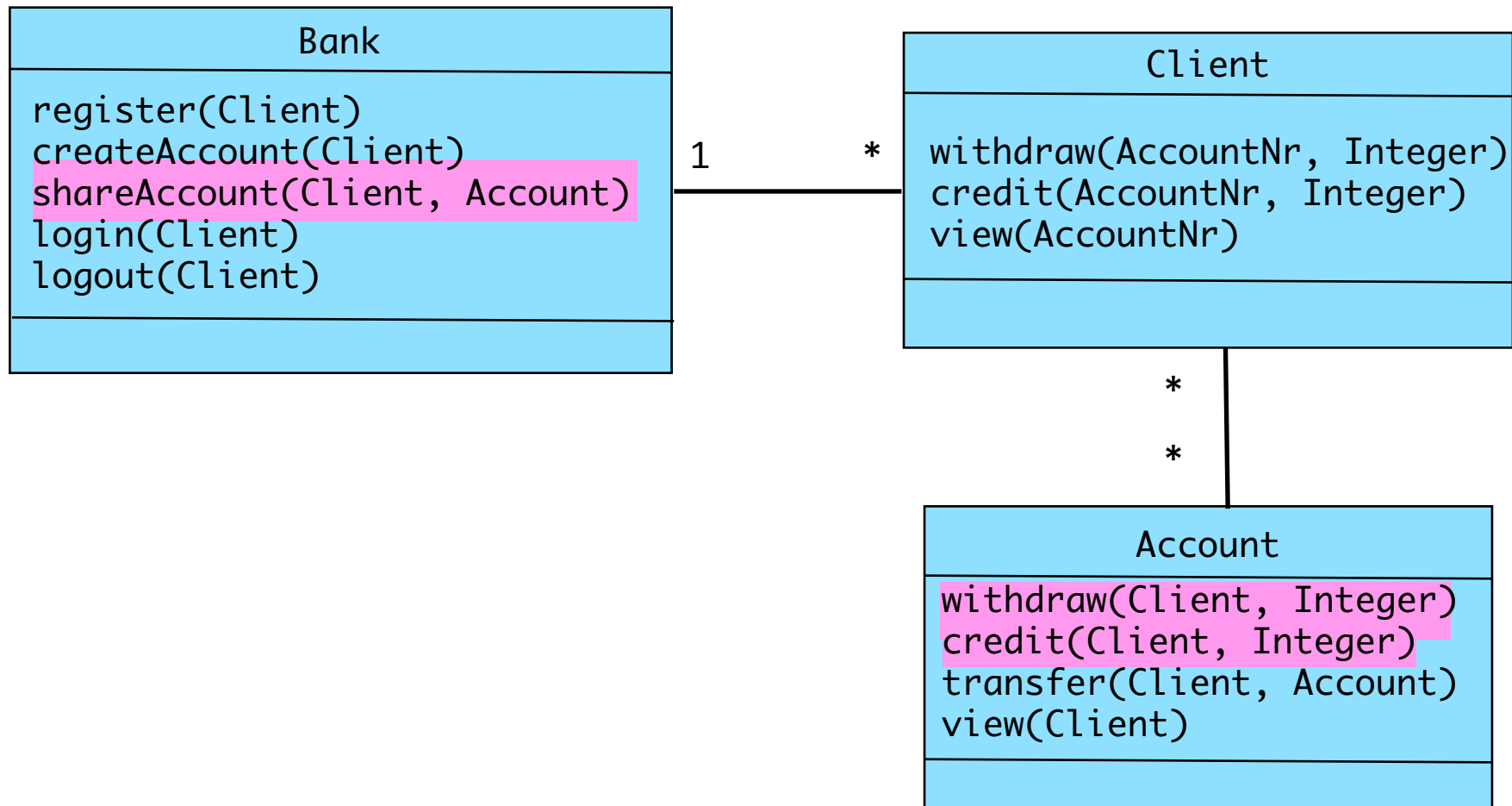
tangling & scattering

- maintainability
- reusability
- readability

CLOS: before, after, around



Aspect-Oriented Programming



# Scattering & tangling

log sensitive operations = crosscutting concern

Tyranny of the dominant decomposition



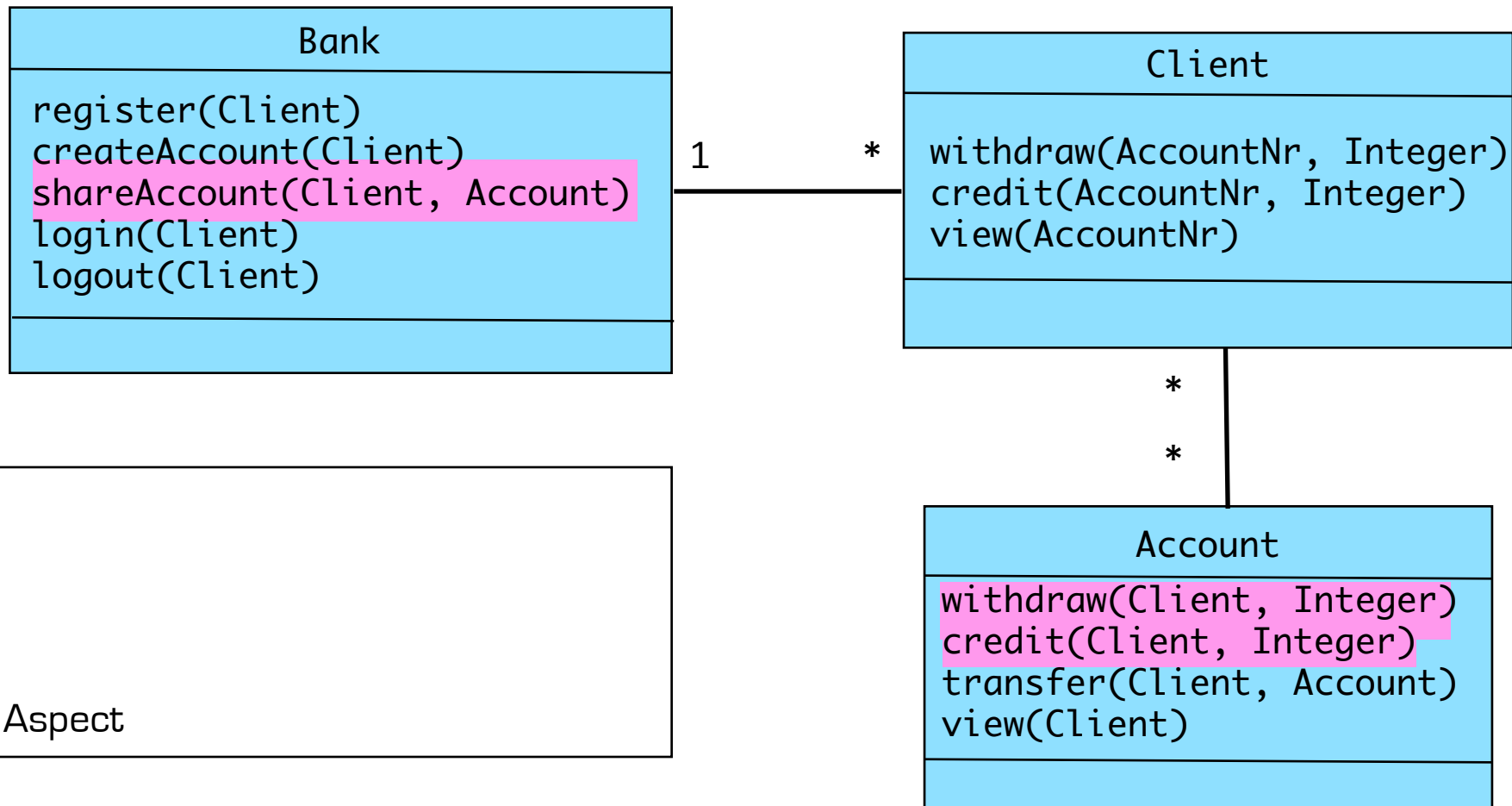
tangling & scattering

- maintainability
- reusability
- readability

CLOS: before, after, around



Aspect-Oriented Programming



# Scattering & tangling

log sensitive operations = crosscutting concern

Tyranny of the dominant decomposition



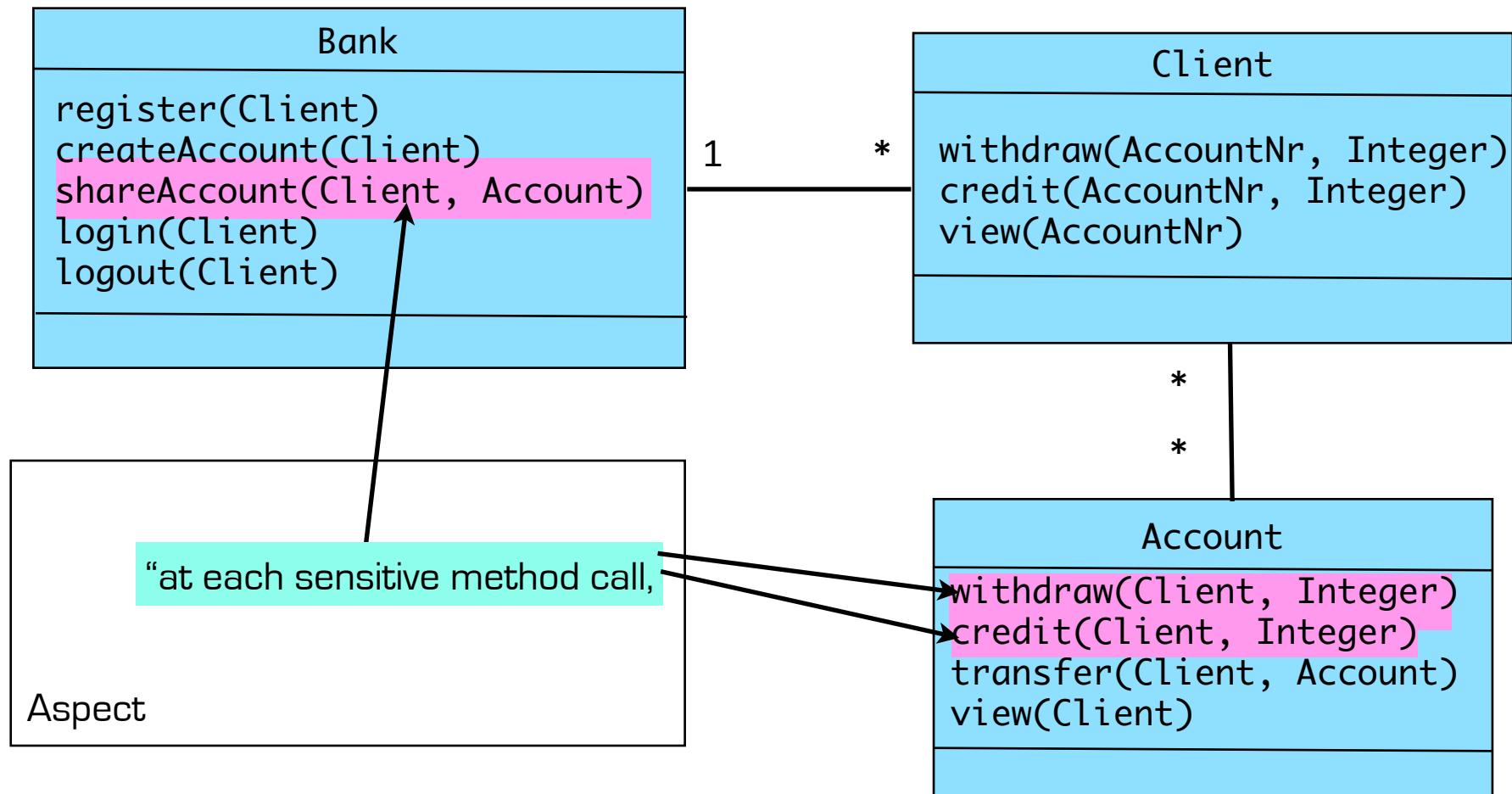
tangling & scattering

- maintainability
- reusability
- readability

CLOS: before, after, around



Aspect-Oriented Programming



# Scattering & tangling

log sensitive operations = crosscutting concern

Tyranny of the dominant decomposition



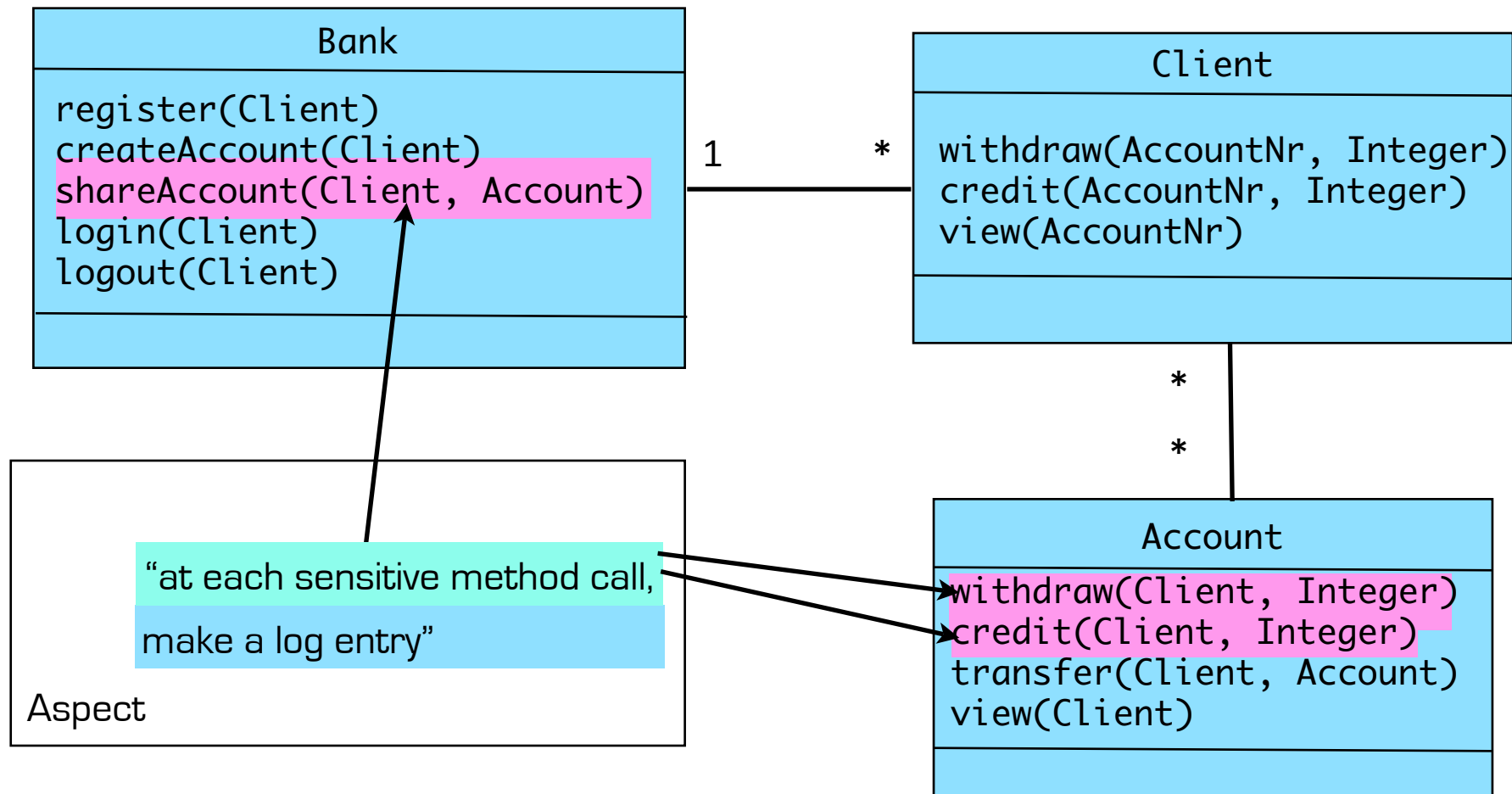
tangling & scattering

- maintainability
- reusability
- readability

CLOS: before, after, around



Aspect-Oriented Programming



# Scattering & tangling

log sensitive operations = crosscutting concern

Tyranny of the dominant decomposition



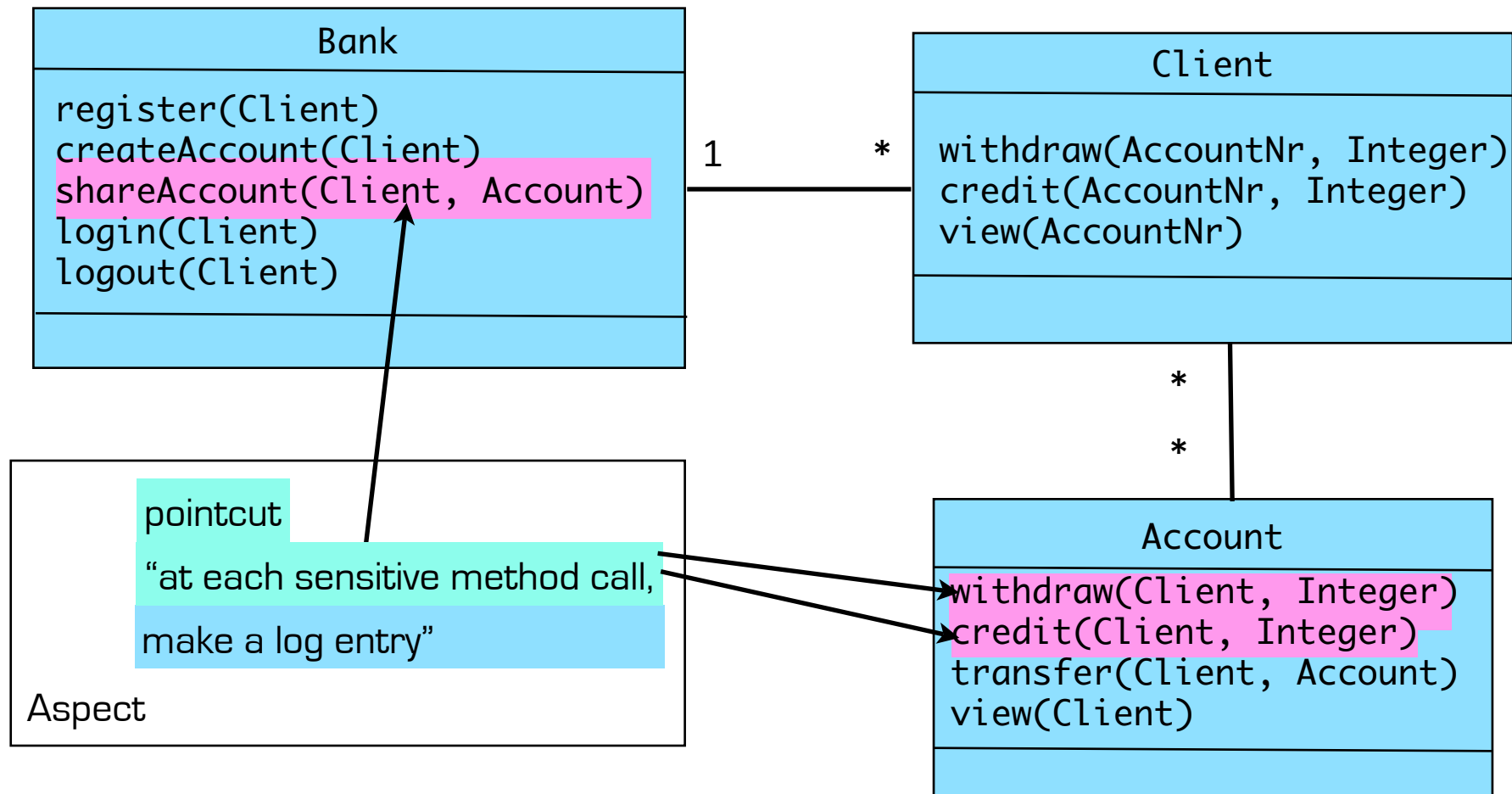
tangling & scattering

- maintainability
- reusability
- readability

CLOS: before, after, around



Aspect-Oriented Programming





# Scattering & tangling

log sensitive operations = crosscutting concern

Tyranny of the dominant decomposition



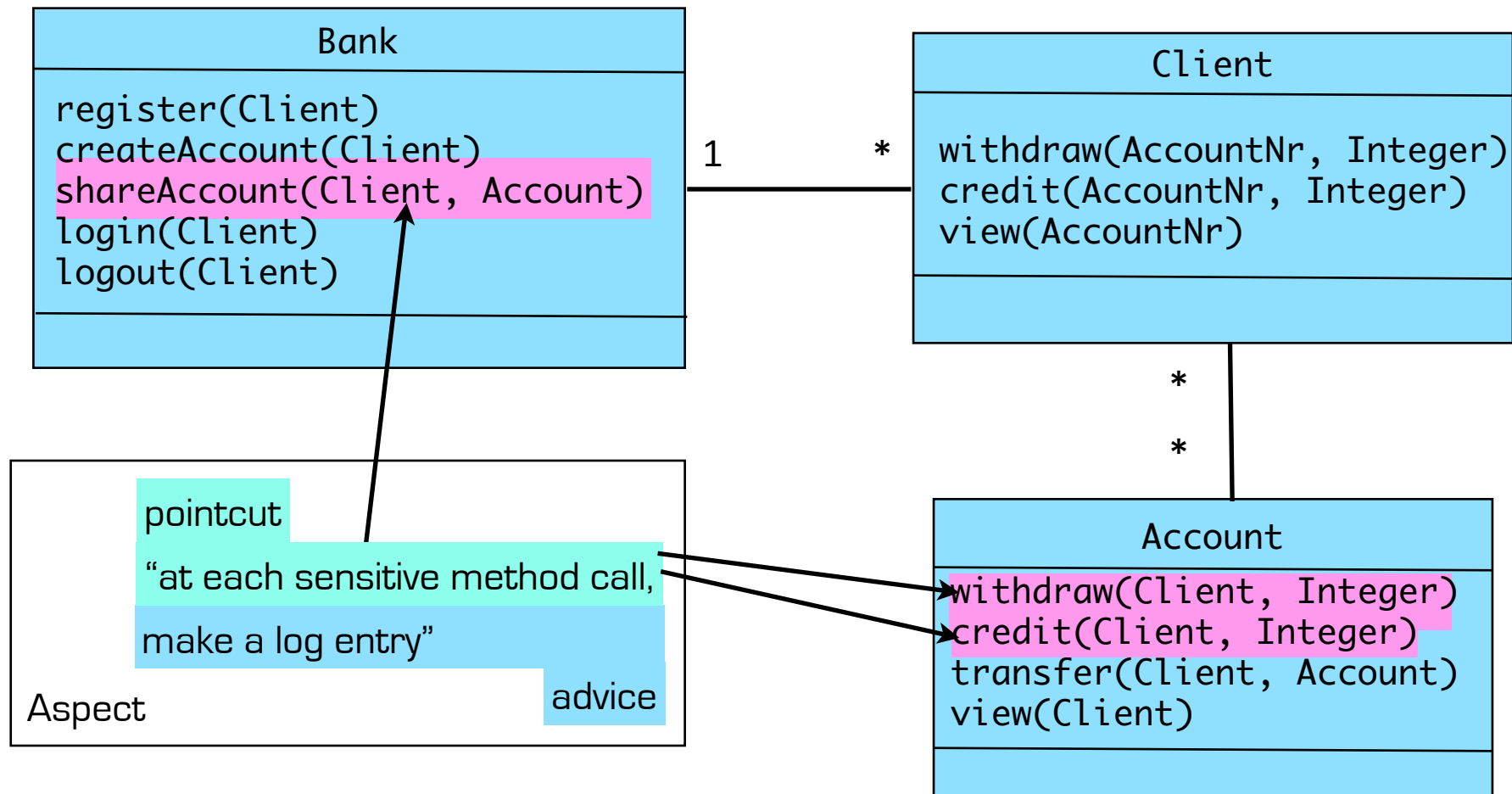
tangling & scattering

- maintainability
- reusability
- readability

CLOS: before, after, around



Aspect-Oriented Programming



# Scattering & tangling

log sensitive operations = crosscutting concern

Tyranny of the dominant decomposition



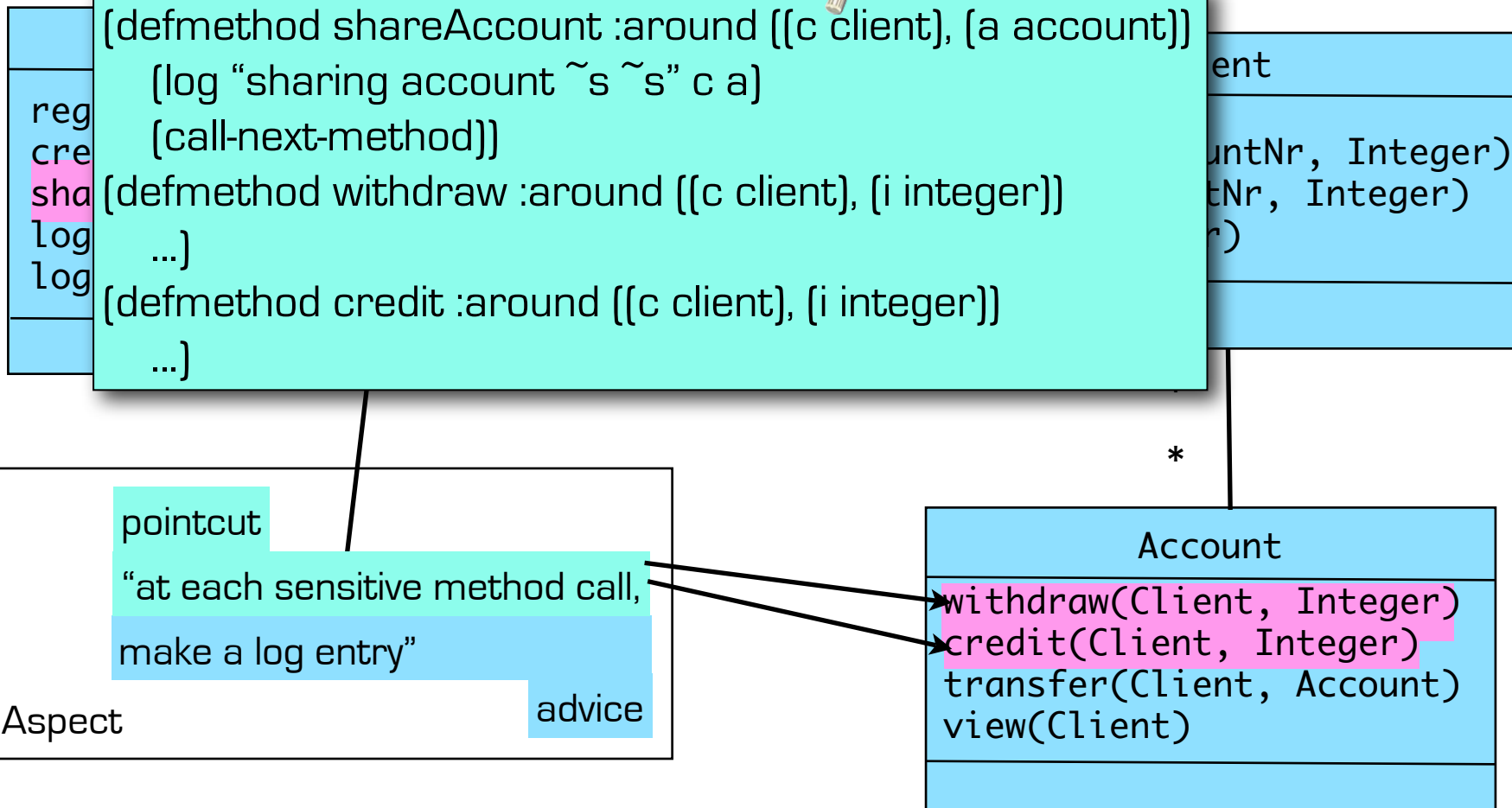
tangling & scattering

- maintainability
- reusability
- readability

CLOS: before, after, around



Aspect-Oriented Programming



# Scattering & tangling

log sensitive operations = crosscutting concern

Tyranny of the dominant decomposition



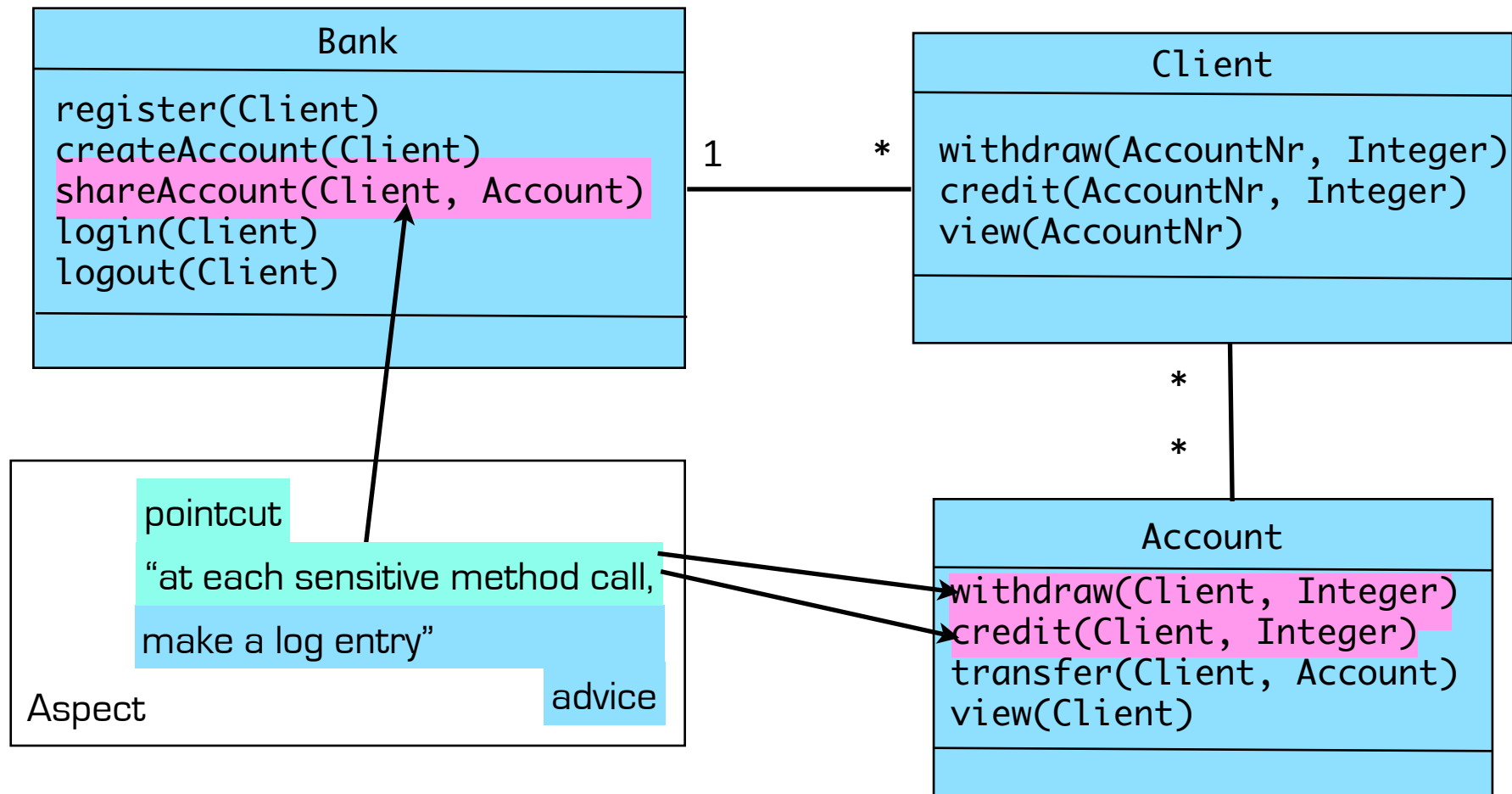
tangling & scattering

- maintainability
- reusability
- readability

CLOS: before, after, around



Aspect-Oriented Programming



# Scattering & tangling

log sensitive operations = crosscutting concern

Tyranny of the dominant decomposition



tangling & scattering

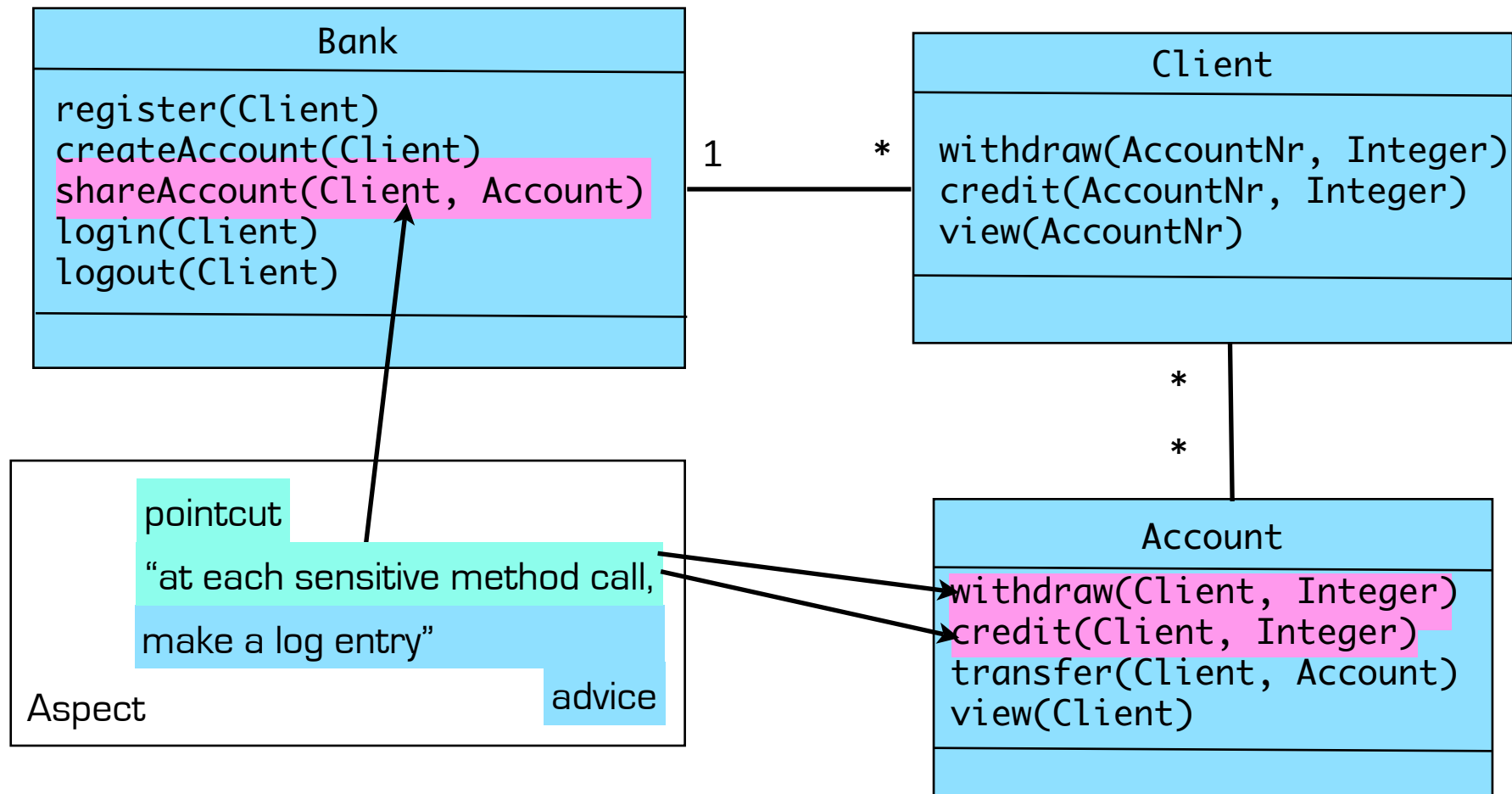
- maintainability
- reusability
- readability

CLOS: before, after, around



?macros

Aspect-Oriented Programming



# Scattering & tangling

log sensitive operations = crosscutting concern

Tyranny of the dominant decomposition



tangling & scattering

- maintainability
- reusability
- readability

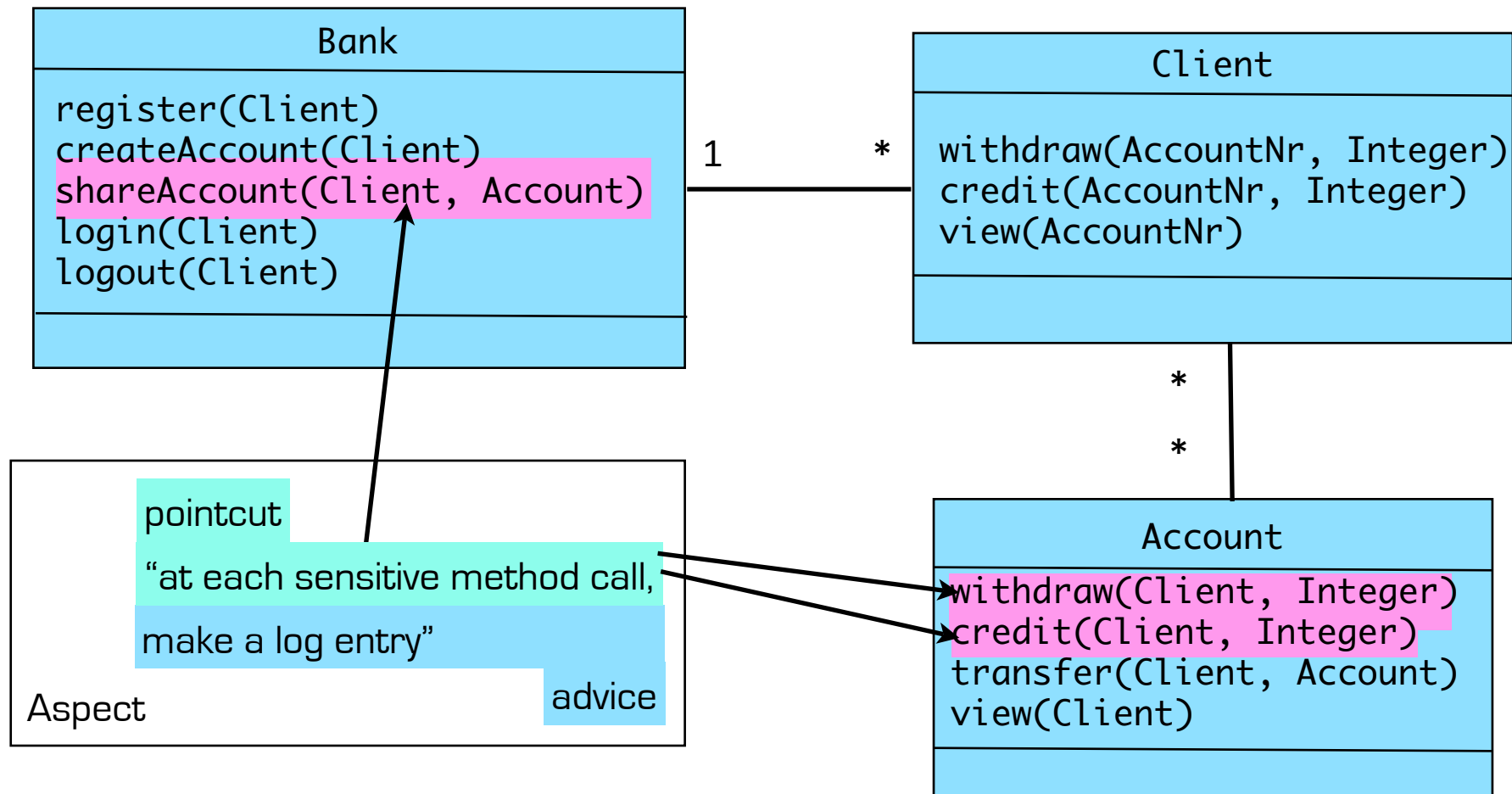
CLOS: before, after, around



?macros

- current join point only

Aspect-Oriented Programming



# Scattering & tangling

log sensitive operations = crosscutting concern

Tyranny of the dominant decomposition



tangling & scattering

- maintainability
- reusability
- readability

CLOS: before, after, around

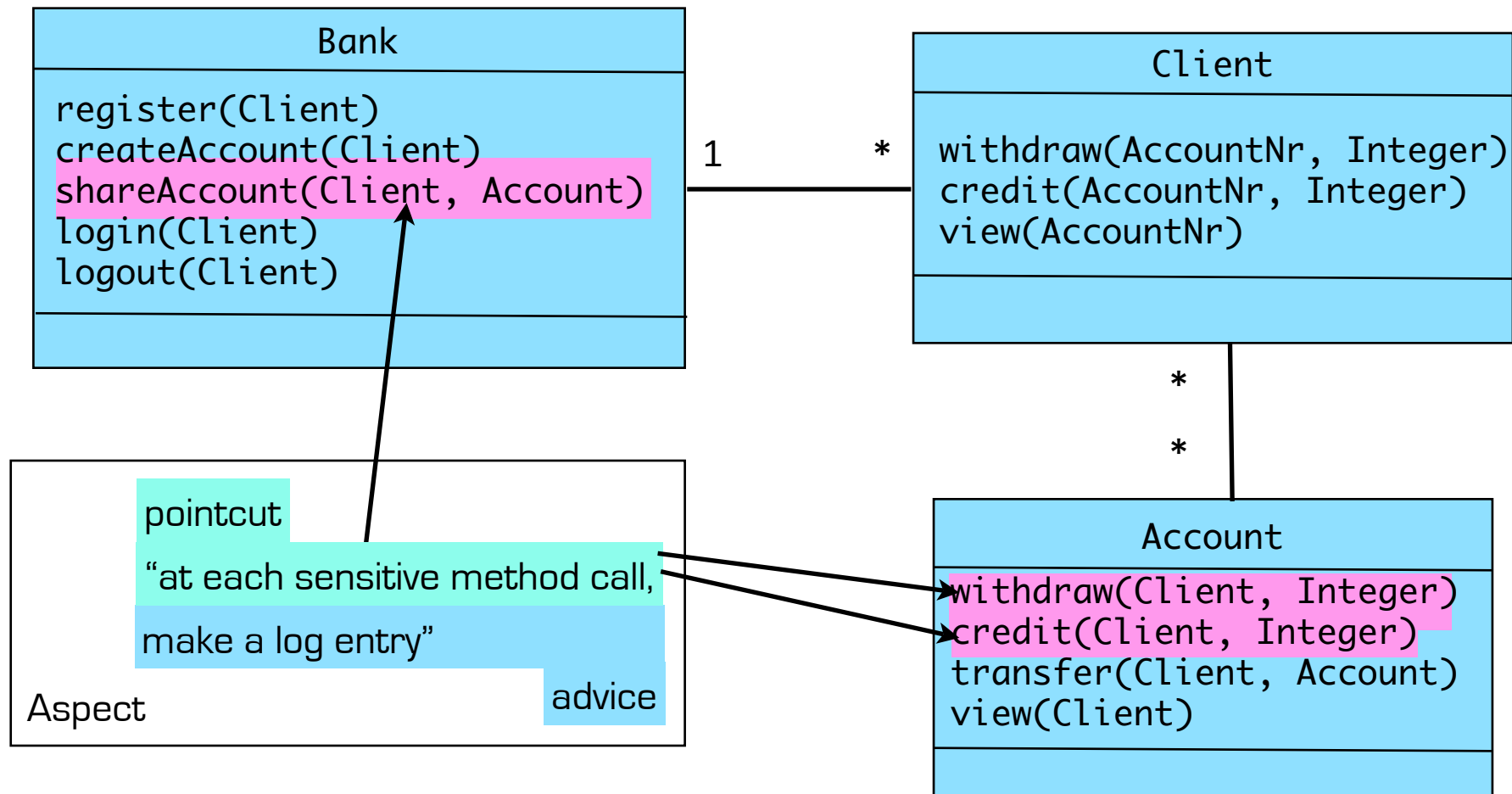


?macros

- current join point only

! history of join points

Aspect-Oriented Programming



# Declarative Meta Programming & Aspect-Oriented Programming

base]

# Declarative Meta Programming & Aspect-Oriented Programming

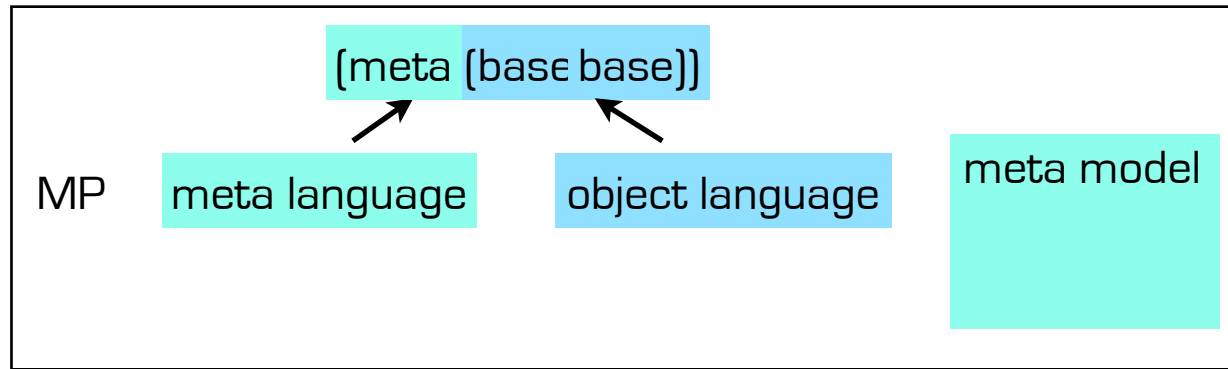
(base base)



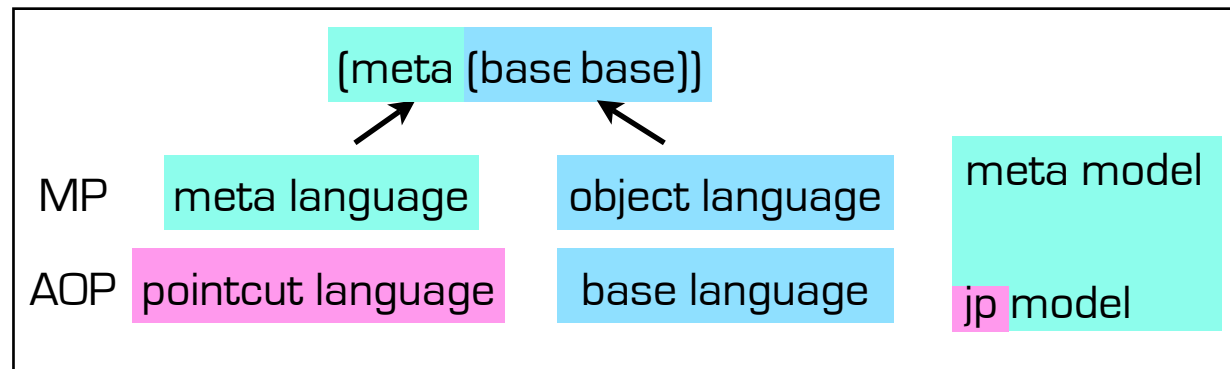
# Declarative Meta Programming & Aspect-Oriented Programming

```
[meta (base base)]
```

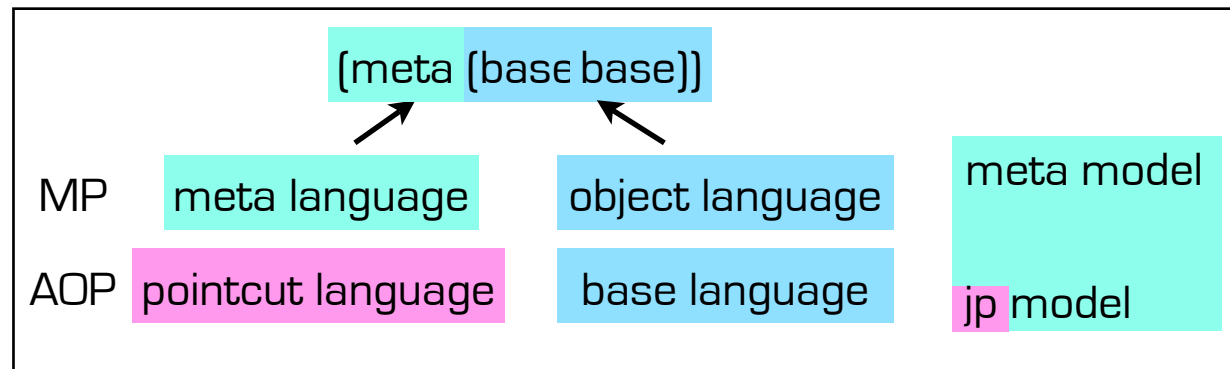
# Declarative Meta Programming & Aspect-Oriented Programming



# Declarative Meta Programming & Aspect-Oriented Programming



# Declarative Meta Programming & Aspect-Oriented Programming



SOUL for DMP:

Base code to facts -> Image querying

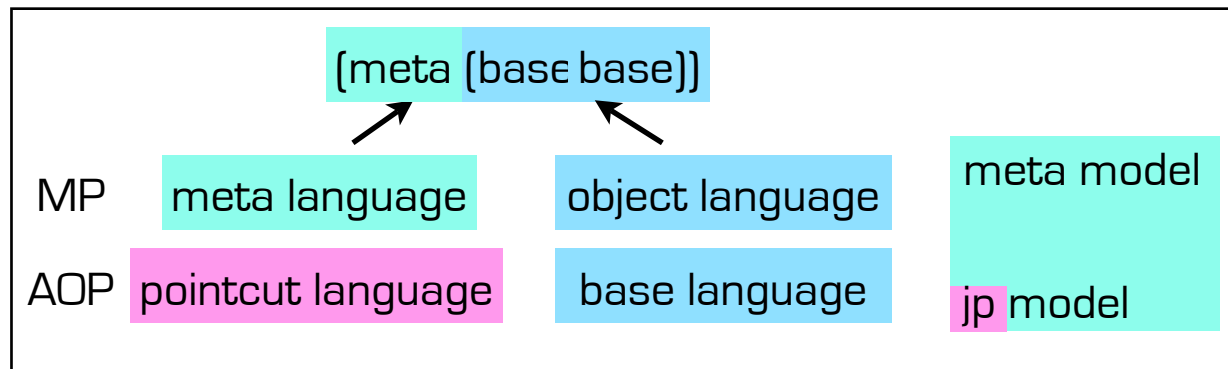
Symbiosis (quasiquoteing)

Rule abstraction (Libraries)

Generating code

**Detecting/Verifying coding patterns**

# Declarative Meta Programming & Aspect-Oriented Programming



SOUL for DMP:

Base code to facts -> Image querying

Symbiosis (quasiquoteing)

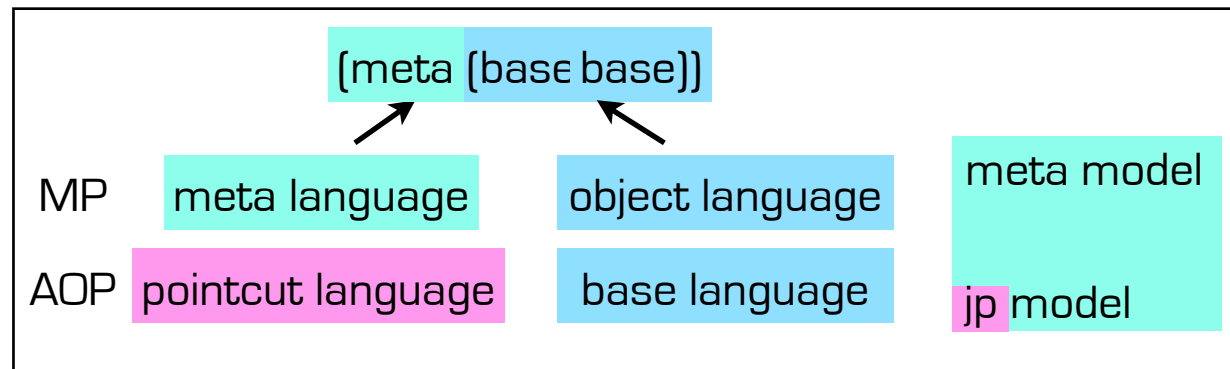
Rule abstraction (Libraries)

Generating code

**Detecting/Verifying coding patterns**

```
(defclass account () ((balance)))  
(defclass client () ((id)))  
(defmethod withdraw ((a account) nr)  
  (setf (balance a) (- (balance a) nr)))  
(defmethod credit ((a account) nr)  
  ...)
```

# Declarative Meta Programming & Aspect-Oriented Programming



SOUL for DMP:

Base code to facts -> Image querying

Symbiosis (quasiquoteing)

Rule abstraction (Libraries)

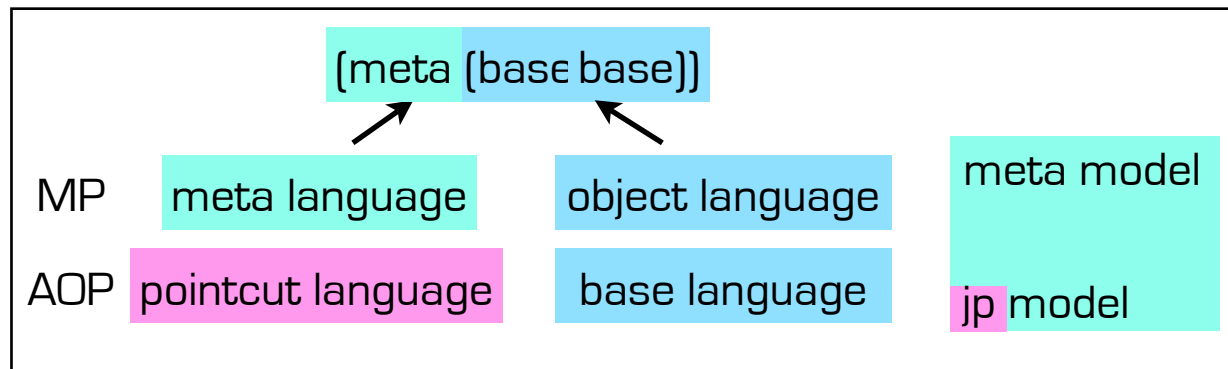
Generating code

**Detecting/Verifying coding patterns**

```
(defclass account () ((balance)))  
(defclass client () ((id)))  
(defmethod withdraw ((a account) nr)  
  (setf (balance a) (- (balance a) nr)))  
(defmethod credit ((a account) nr)  
  ...)
```



# Declarative Meta Programming & Aspect-Oriented Programming



SOUL for DMP:

Base code to facts -> Image querying

Symbiosis (quasiquoting)

Rule abstraction (Libraries)

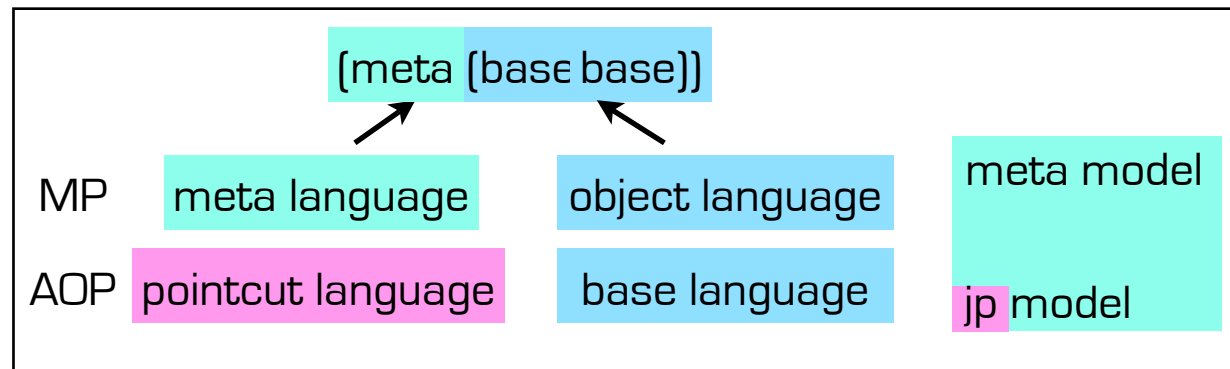
Generating code

**Detecting/Verifying coding patterns**

```
(defclass account () ((balance)))  
(defclass client () ((id)))  
(defmethod withdraw ((a account) nr)  
  (setf (balance a) (- (balance a) nr)))  
(defmethod credit ((a account) nr)  
  ...)
```

```
class( 'account , <account>)  
classImplementsMethodNamed  
( <account> , 'withdraw, <withdraw>)  
method( <account> , <withdraw>)  
methodName( <withdraw> , 'withdraw)  
methodArguments( <withdraw> , ())  
methodStatements( <withdraw> , ((setf  
  (balance a) (- (balance a) nr))))  
...
```

# Declarative Meta Programming & Aspect-Oriented Programming



SOUL for DMP:

Base code to facts -> Image querying

Symbiosis (quasiquoteing)

Rule abstraction (Libraries)

Generating code

**Detecting/Verifying coding patterns**

```
(defclass account () ((balance)))  
(defclass client () ((id)))  
(defmethod withdraw ((a account) nr)  
  (setf (balance a) (- (balance a) nr)))  
(defmethod credit ((a account) nr)  
  ...)
```

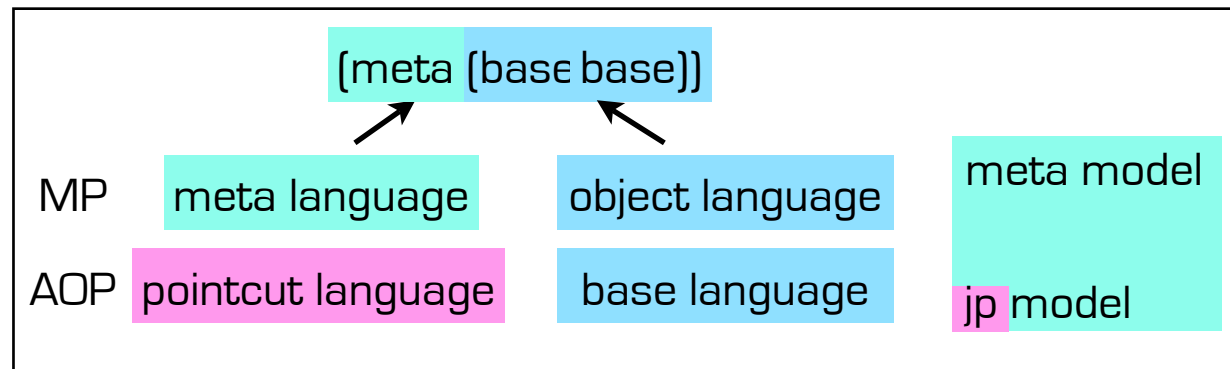


```
abstractClassHeuristic() if  
forall(abstractClass(?C),baseClass(?C)),  
forall(baseClass(?C),abstractClass(?C)).
```

```
class( 'account , <account> )  
classImplementsMethodNamed  
( <account> , 'withdraw , <withdraw> )  
method( <account> , <withdraw> )  
methodName( <withdraw> , 'withdraw )  
methodArguments( <withdraw> , () )  
methodStatements( <withdraw> , ((setf  
  (balance a) (- (balance a) nr))) )  
...
```



# Declarative Meta Programming & Aspect-Oriented Programming



SOUL for DMP:

Base code to facts -> Image querying

Symbiosis (quasiquoteing)

Rule abstraction (Libraries)

Generating code

**Detecting/Verifying coding patterns**

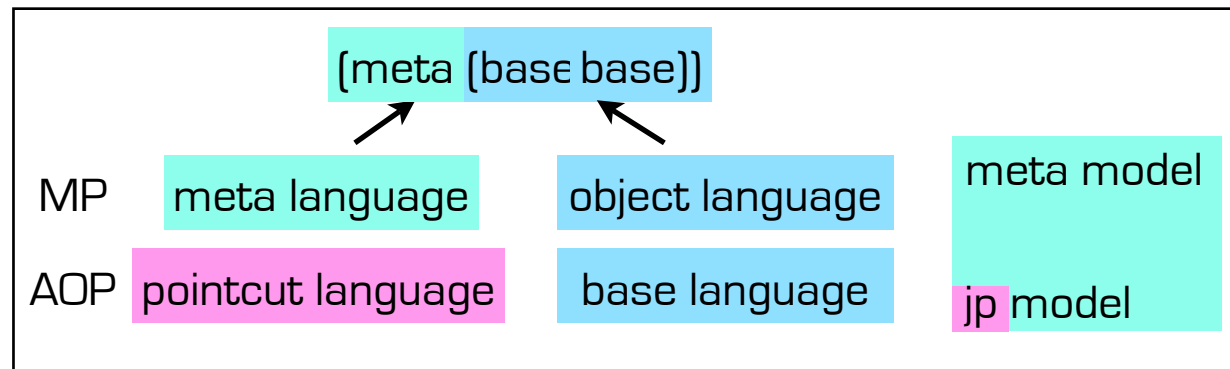
```
(defclass account () ((balance)))  
(defclass client () ((id)))  
(defmethod withdraw ((a account) nr)  
  (setf (balance a) (- (balance a) nr)))  
(defmethod credit ((a account) nr)  
  ...)
```

```
abstractClassHeuristic() if  
forall(abstractClass(?C),baseClass(?C)),  
forall(baseClass(?C),abstractClass(?C)).
```

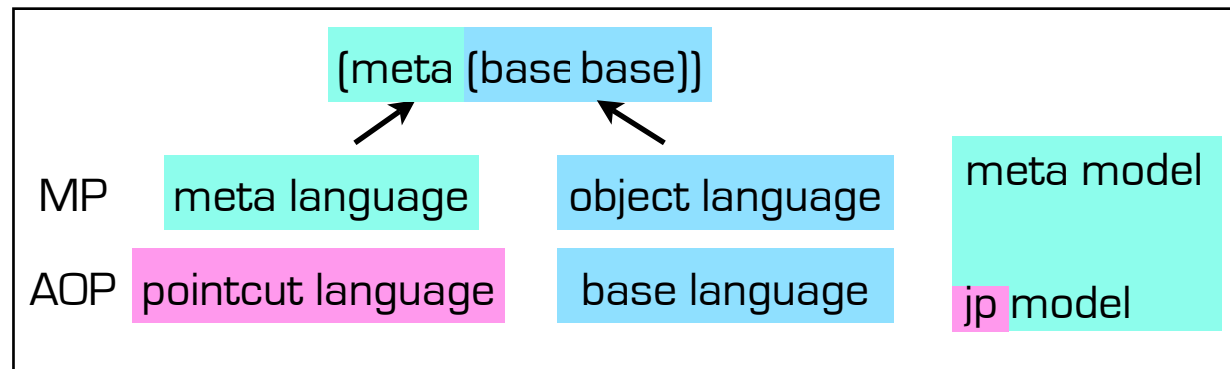
```
class( 'account , <account>)  
classImplementsMethodNamed  
( <account> , 'withdraw, <withdraw>)  
method(<account>, <withdraw>)  
methodName(<withdraw>, 'withdraw)  
methodArguments(<withdraw>, ())  
methodStatements(<withdraw>, ((setf  
  (balance a) (- (balance a) nr))))  
...
```

compute

# Declarative Meta Programming & Aspect-Oriented Programming

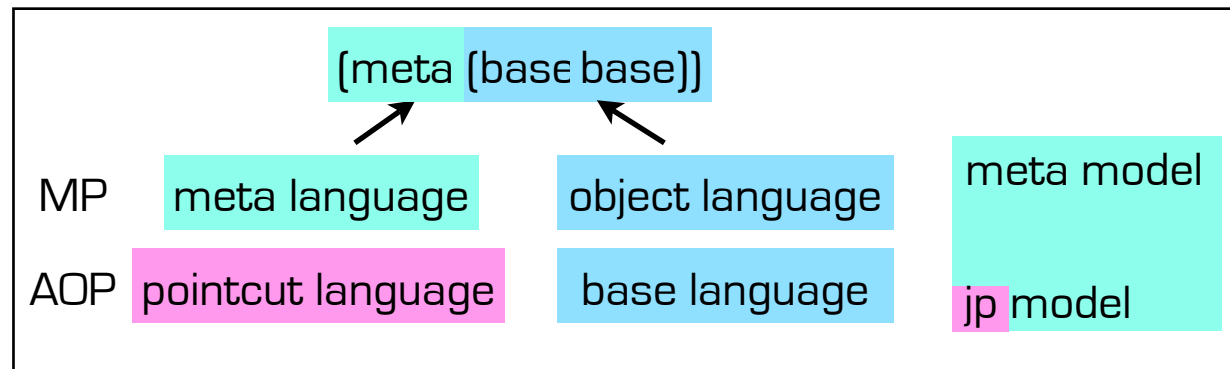


# Declarative Meta Programming & Aspect-Oriented Programming



CARMA: AOP with a SOUL  
modularizing scattered & tangled code  
Join point type predicates  
Link to shadows  
Symbiosis  
Loose coupling aspect/base  
**Pattern-based pointcuts**

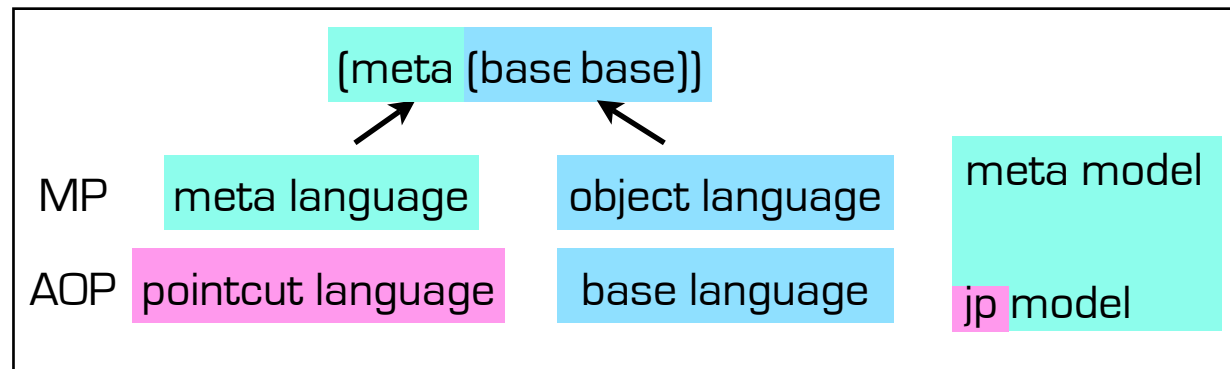
# Declarative Meta Programming & Aspect-Oriented Programming



CARMA: AOP with a SOUL  
modularizing scattered & tangled code  
Join point type predicates  
Link to shadows  
Symbiosis  
Loose coupling aspect/base  
**Pattern-based pointcuts**

```
(defclass account () ...)
(defmethod withdraw ((a account) nr)
  (setf (balance a) (- (balance a) nr)))
(defmethod credit ((a account) nr) ...)
(withdraw *account* 3)
(print (get-value *account* 'balance))
```

# Declarative Meta Programming & Aspect-Oriented Programming

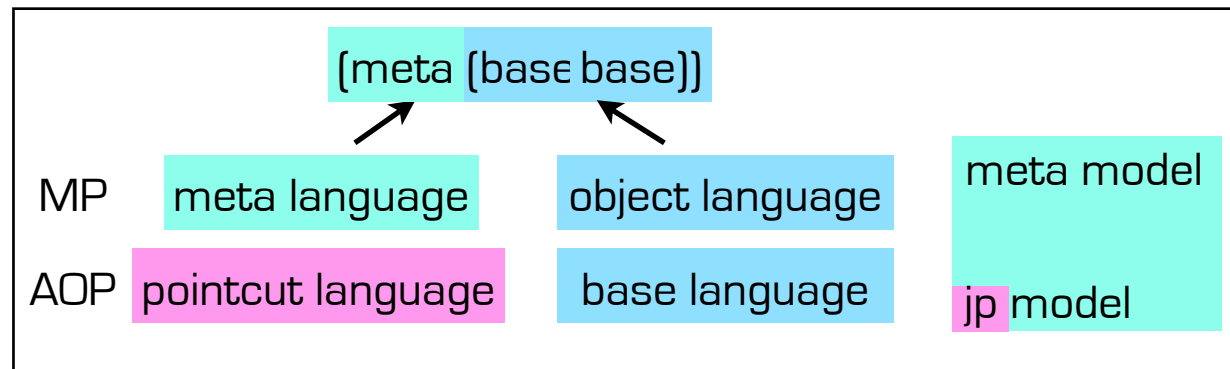


CARMA: AOP with a SOUL  
modularizing scattered & tangled code  
Join point type predicates  
Link to shadows  
Symbiosis  
Loose coupling aspect/base  
**Pattern-based pointcuts**

```
(defclass account ())  
(defmethod withdraw ((a account) nr)  
  (setf (balance a) (- (balance a) nr)))  
(defmethod credit ((a account) nr) ...)  
(withdraw *account* 3)  
(print (get-value *account* 'balance))
```



# Declarative Meta Programming & Aspect-Oriented Programming

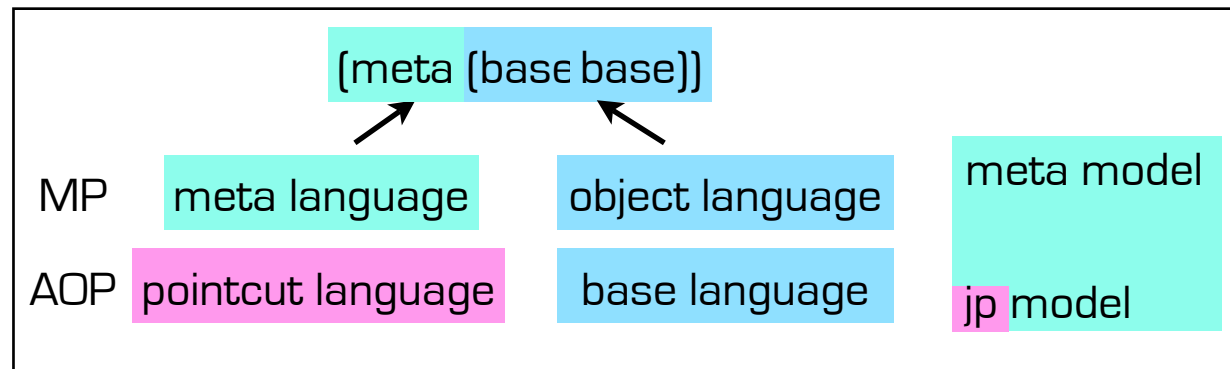


CARMA: AOP with a SOUL  
modularizing scattered & tangled code  
Join point type predicates  
Link to shadows  
Symbiosis  
Loose coupling aspect/base  
**Pattern-based pointcuts**

```
(defclass account () ...)  
(defmethod withdraw ((a account) nr)  
  (setf (balance a) (- (balance a) nr)))  
(defmethod credit ((a account) nr) ...)  
(withdraw *account* 3)  
(print (get-value *account* 'balance))
```

```
class( 'account , <account>)  
classImplementsMethodNamed  
(<account> , 'withdraw, <withdraw>)  
1. send(<jp>, withdraw, [<account>])  
   assignment(<jp>, balance, 10, 7)  
2. reference(<jp2>, balance, 7)
```

# Declarative Meta Programming & Aspect-Oriented Programming



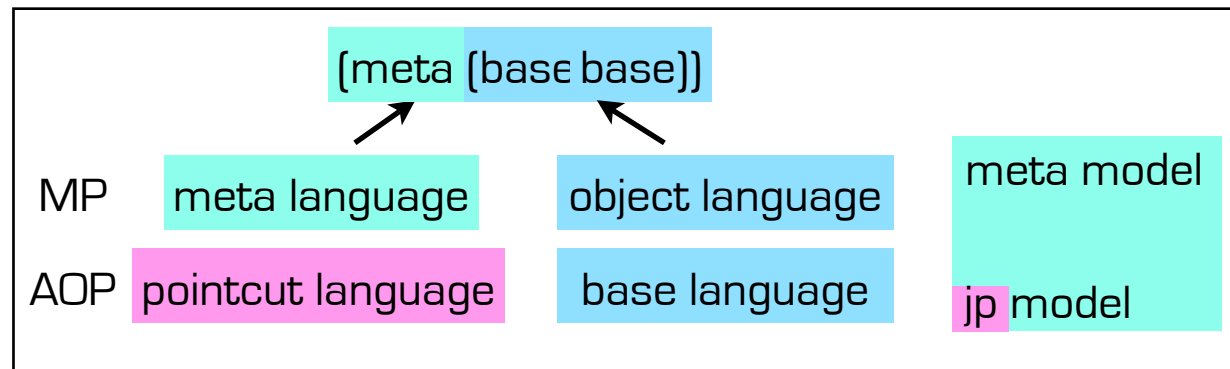
CARMA: AOP with a SOUL  
modularizing scattered & tangled code  
Join point type predicates  
Link to shadows  
Symbiosis  
Loose coupling aspect/base  
**Pattern-based pointcuts**

carma pointcut

```
(defclass account () ...)
(defmethod withdraw ((a account) nr)
  (setf (balance a) (- (balance a) nr)))
(defmethod credit ((a account) nr) ...)
(withdraw *account* 3)
(print (get-value *account* 'balance))
```

```
class( 'account , <account>)
classImplementsMethodNamed
(<account> , 'withdraw, <withdraw>)
1. send(<jp>, withdraw, [<account>])
   assignment(<jp>, balance, 10, 7)
2. reference(<jp2>, balance, 7)
```

# Declarative Meta Programming & Aspect-Oriented Programming



CARMA: AOP with a SOUL  
modularizing scattered & tangled code  
Join point type predicates  
Link to shadows  
Symbiosis  
Loose coupling aspect/base  
**Pattern-based pointcuts**

carma pointcut

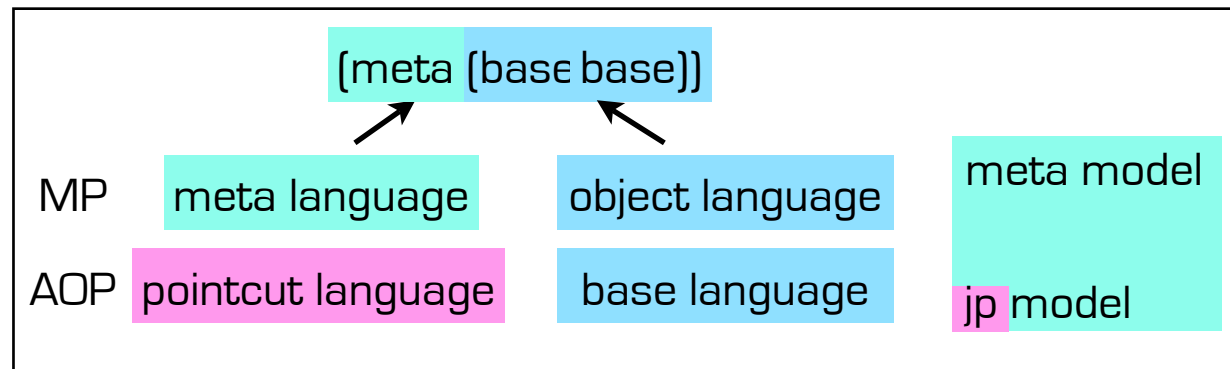
**rule abstraction,  
unification,  
recursion**

```
(defclass account () ...)
(defmethod withdraw ((a account) nr)
  (setf (balance a) (- (balance a) nr)))
(defmethod credit ((a account) nr) ...)
(withdraw *account* 3)
(print (get-value *account* 'balance))
```

```
class( 'account , <account>)
classImplementsMethodNamed
(<account> , 'withdraw, <withdraw>)
1. send(<jp>, withdraw, [<account>])
   assignment(<jp>, balance, 10, 7)
2. reference(<jp2>, balance, 7)
```



# Declarative Meta Programming & Aspect-Oriented Programming



CARMA: AOP with a SOUL  
modularizing scattered & tangled code  
Join point type predicates  
Link to shadows  
Symbiosis  
Loose coupling aspect/base  
**Pattern-based pointcuts**

carma pointcut

rule abstraction,  
unification,  
recursion

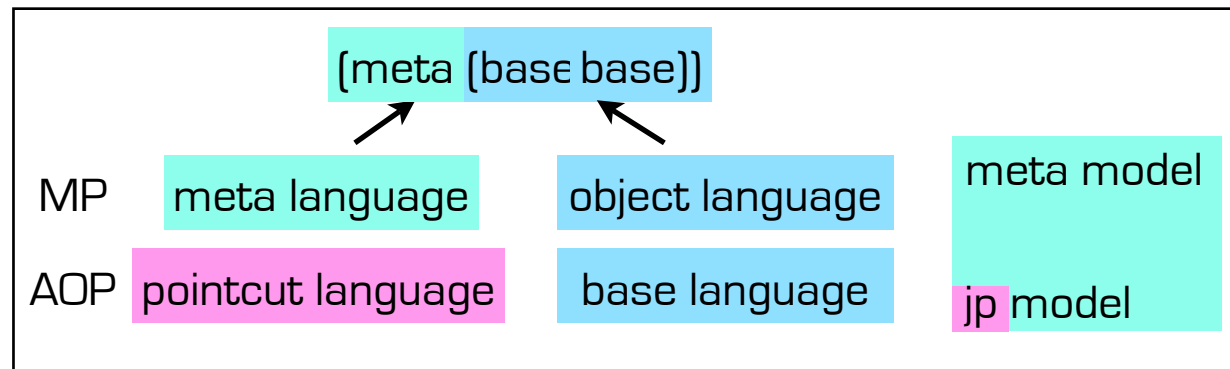
reusability,  
reasoning



```
(defclass account () ...)  
(defmethod withdraw ((a account) nr)  
  (setf (balance a) (- (balance a) nr)))  
(defmethod credit ((a account) nr) ...)  
(withdraw *account* 3)  
(print (get-value *account* 'balance))
```

```
class( 'account , <account>)  
classImplementsMethodNamed  
(<account> , 'withdraw, <withdraw>)  
1. send(<jp>, withdraw, [<account>])  
   assignment(<jp>, balance, 10, 7)  
2. reference(<jp2>, balance, 7)
```

# Declarative Meta Programming & Aspect-Oriented Programming



```
(defclass account () ...)
(defmethod withdraw ((a account) nr)
  (setf (balance a) (- (balance a) nr)))
(defmethod credit ((a account) nr) ...)
(withdraw *account* 3)
(print (get-value *account* 'balance))
```



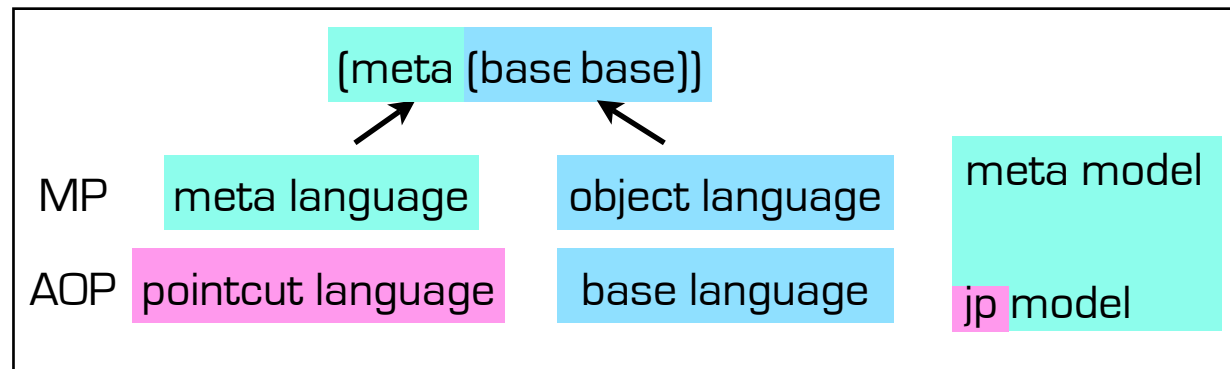
```
class( 'account , <account>)
classImplementsMethodNamed
(<account> , 'withdraw, <withdraw>)
1. send(<jp>, withdraw, [<account>])
   assignment(<jp>, balance, 10, 7)
2. reference(<jp2>, balance, 7)
```

rule abstraction,  
unification,  
recursion

reusability,  
reasoning



# Declarative Meta Programming & Aspect-Oriented Programming



HALO: AOP with a memory  
**Temporal/Stateful pointcuts**

```
(defclass account () ...)
(defmethod withdraw ((a account) nr)
  (setf (balance a) (- (balance a) nr)))
(defmethod credit ((a account) nr) ...)
(withdraw *account* 3)
(print (get-value *account* 'balance))
```

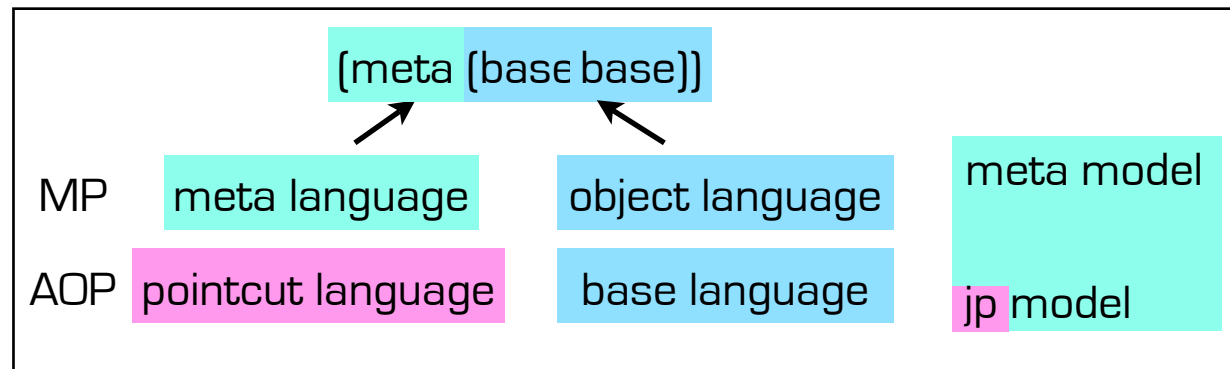
rule abstraction,  
unification,  
recursion

reusability,  
reasoning



```
class( 'account , <account>)
classImplementsMethodNamed
(<account> , 'withdraw, <withdraw>)
1. send(<jp>, withdraw, [<account>])
assignment(<jp>, balance, 10, 7)
2. reference(<jp2>, balance, 7)
```

# Declarative Meta Programming & Aspect-Oriented Programming



HALO: AOP with a memory  
**Temporal/Stateful pointcuts**

```
(defclass account () ...)
(defmethod withdraw ((a account) nr)
  (setf (balance a) (- (balance a) nr)))
(defmethod credit ((a account) nr) ...)
(withdraw *account* 3)
(print (get-value *account* 'balance))
```

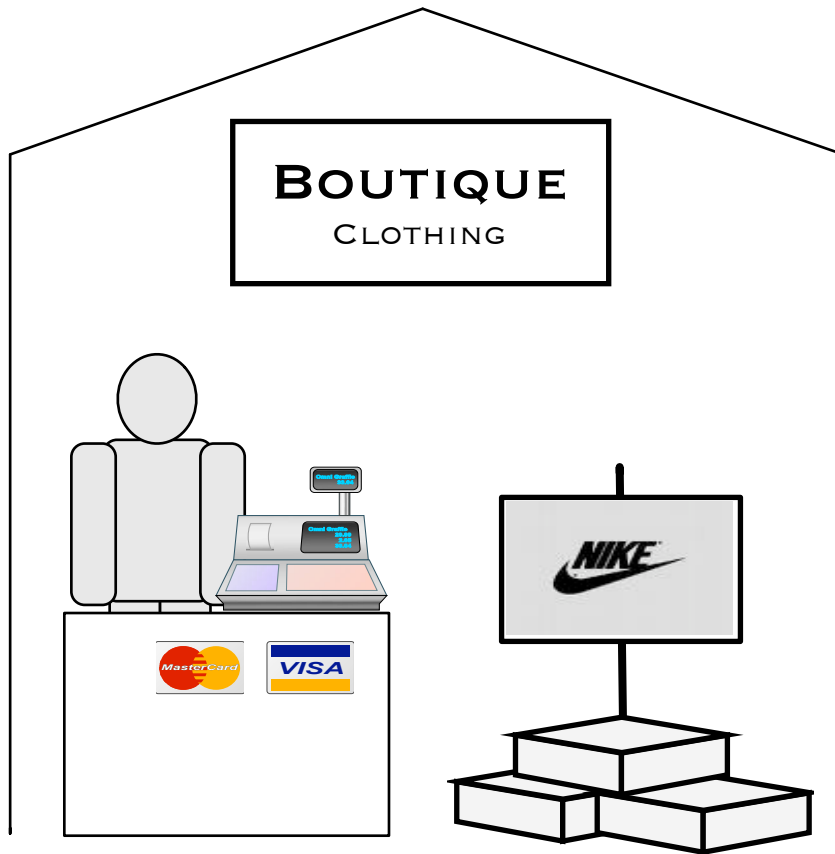
rule abstraction,  
unification,  
recursion

reusability,  
reasoning

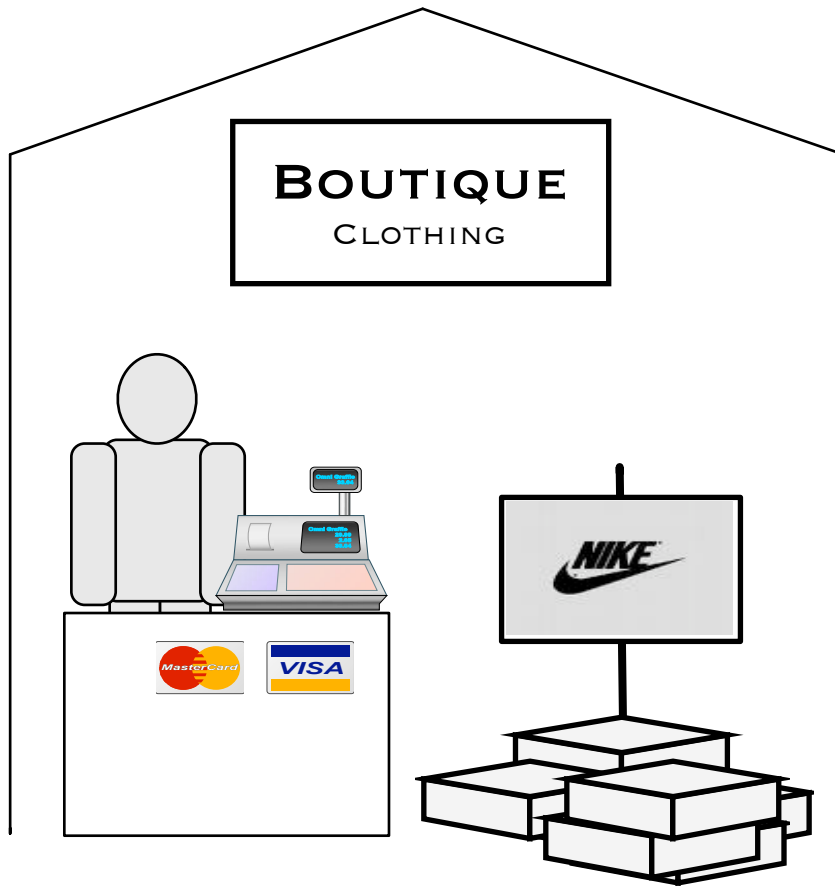


```
class( 'account , <account>) record
classImplementsMethodNamed
(<account> , 'withdraw, <withdraw>)
1. send(<jp>, withdraw, [<account>])
   assignment(<jp>, balance, 10, 7)
2. reference(<jp2>, balance, 7)
```

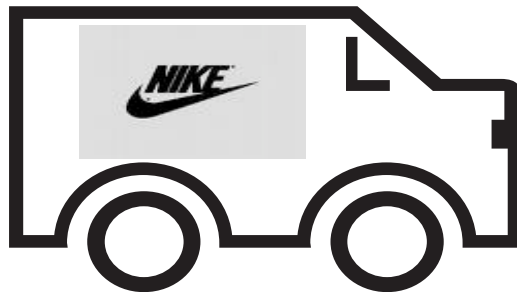
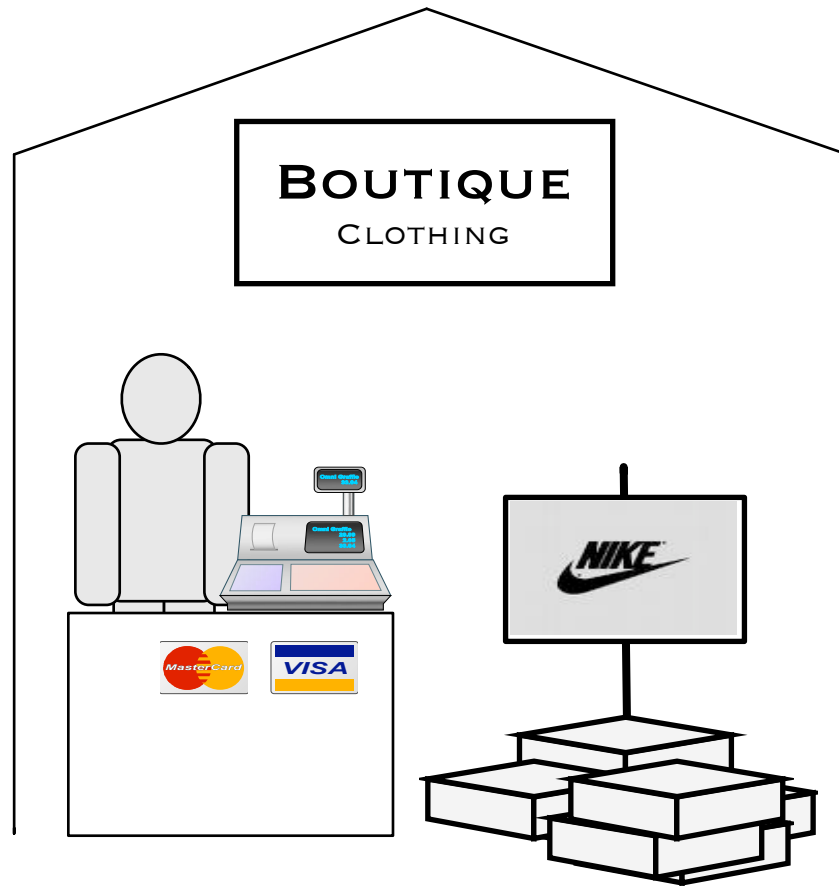
# Running example



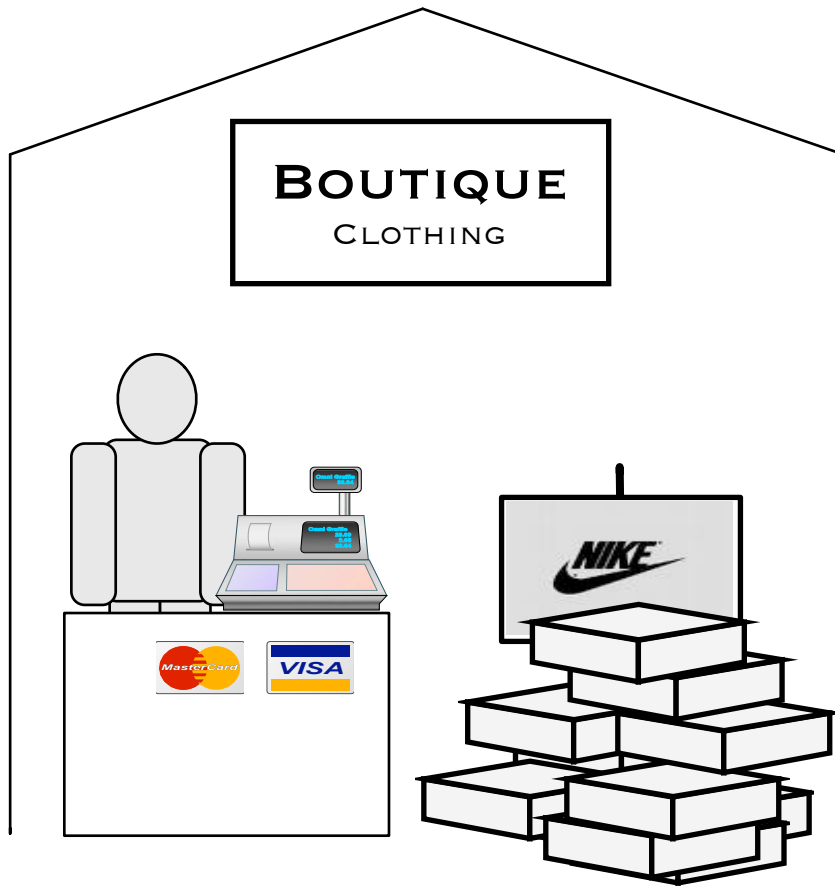
# Running example



# Running example



# Running example

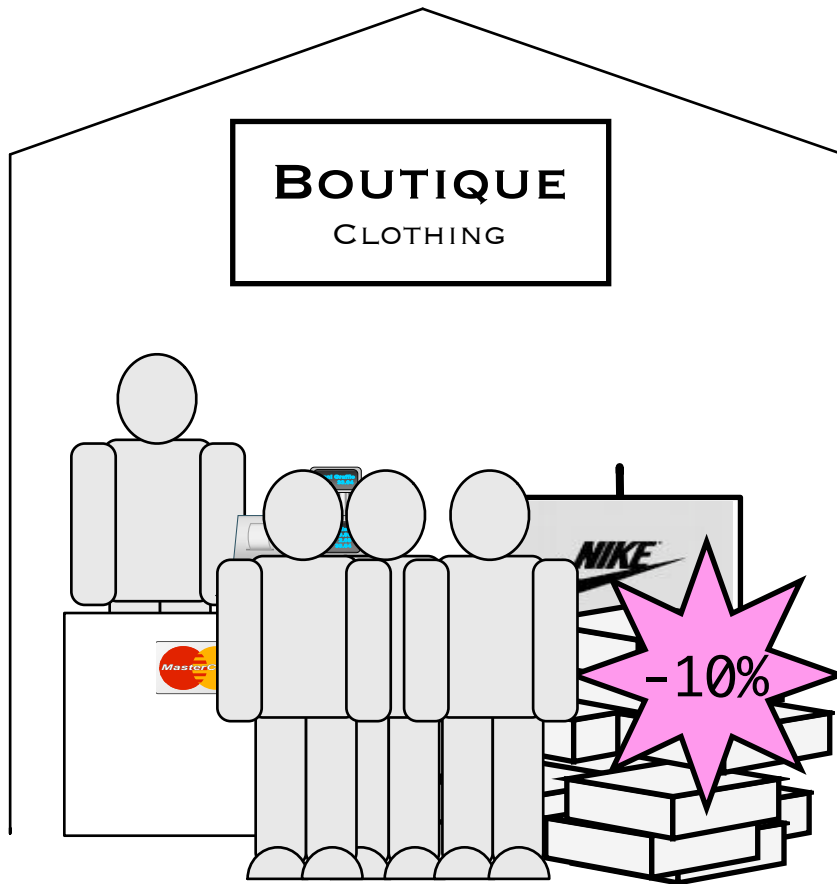




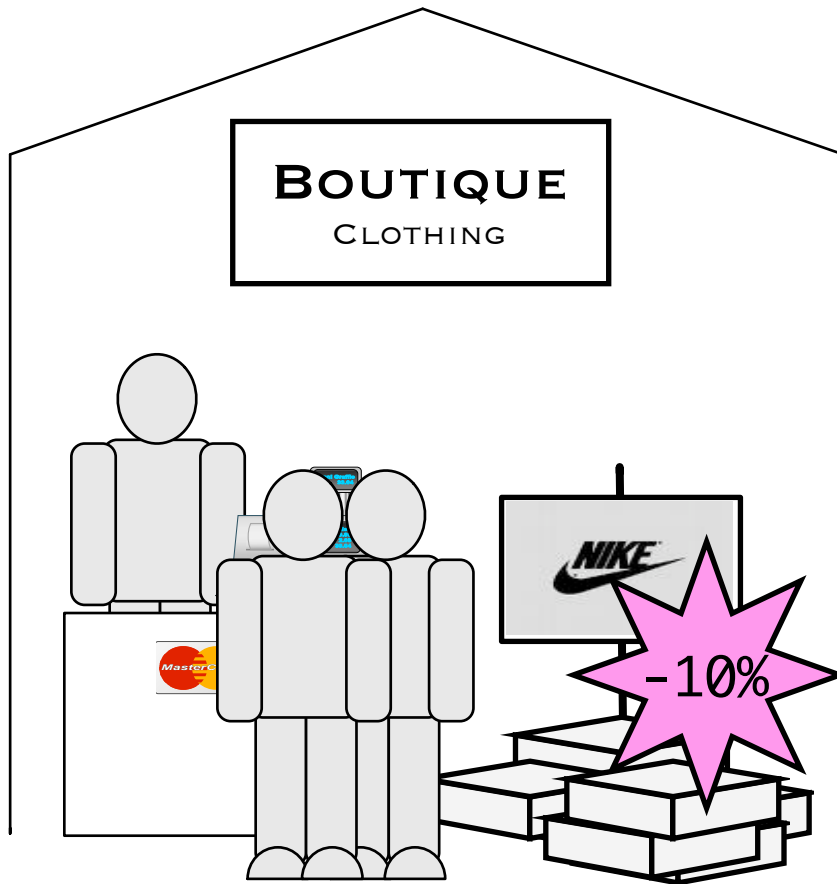
# Running example



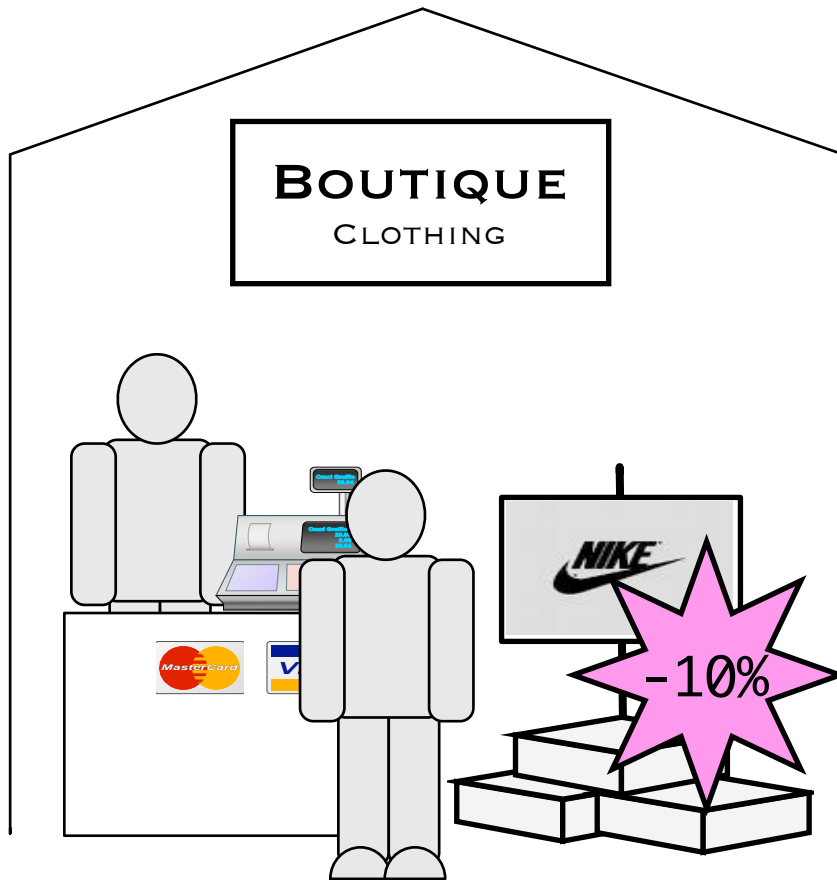
# Running example



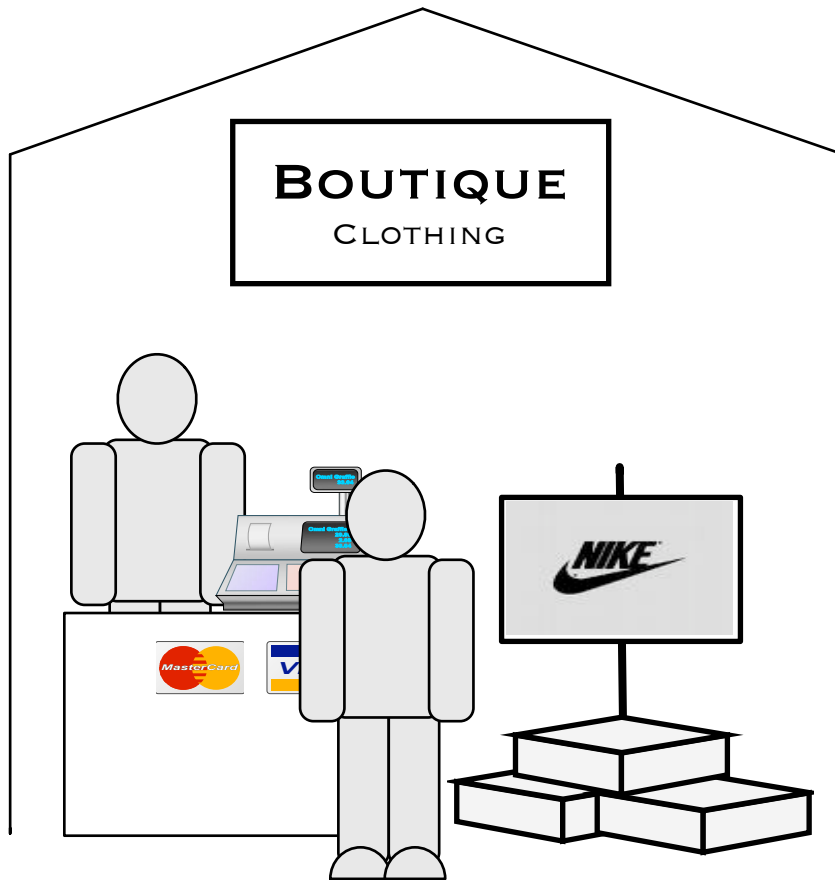
# Running example



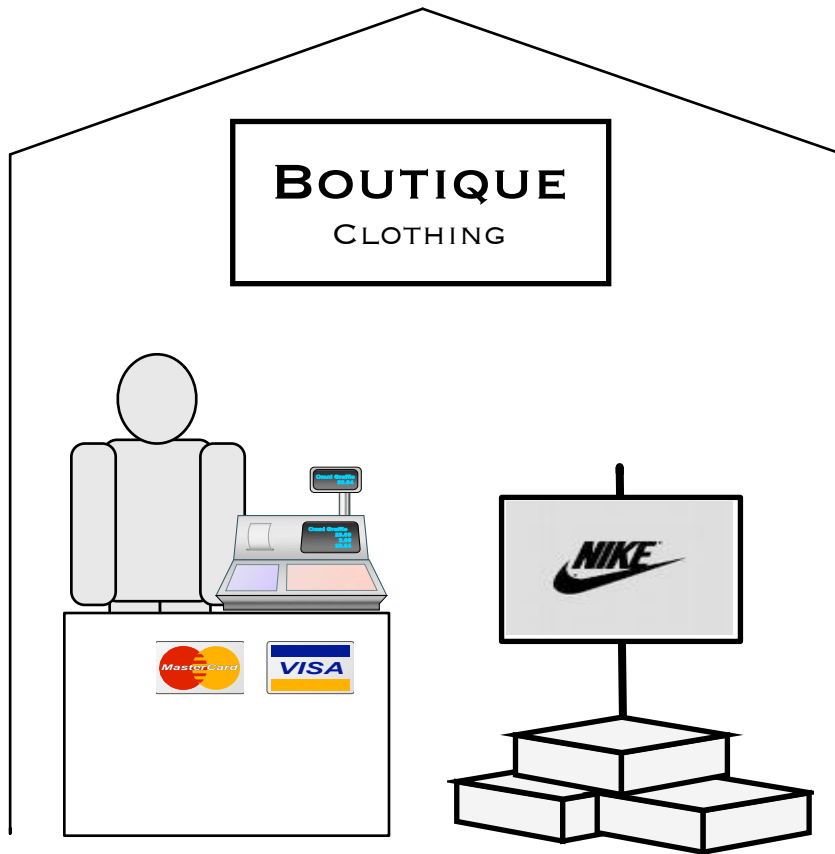
# Running example



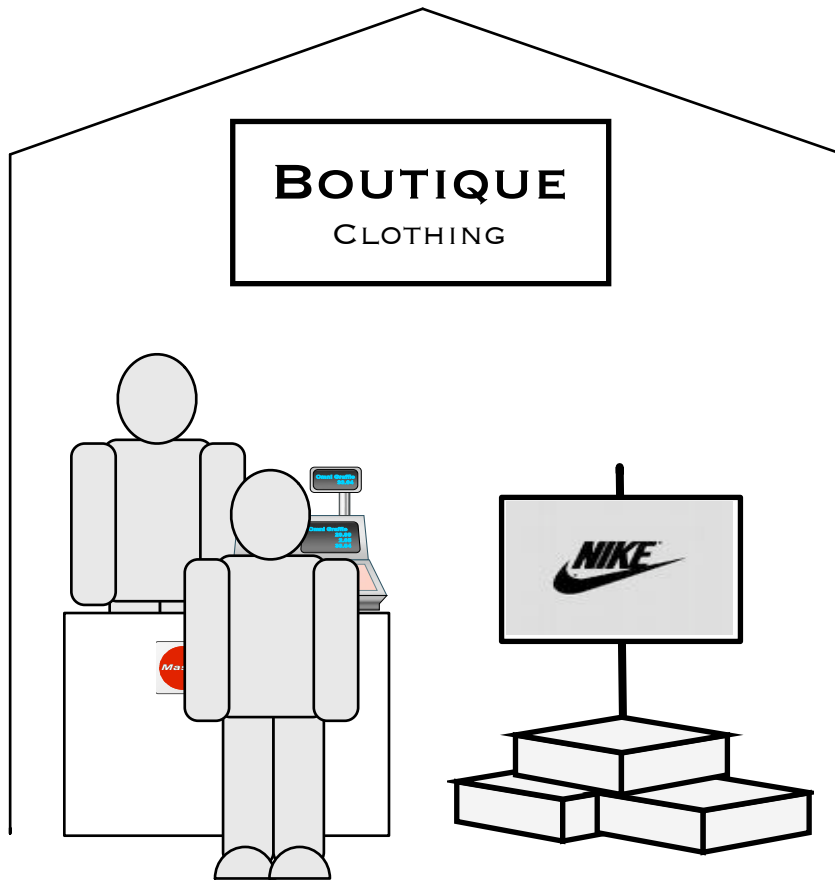
# Running example



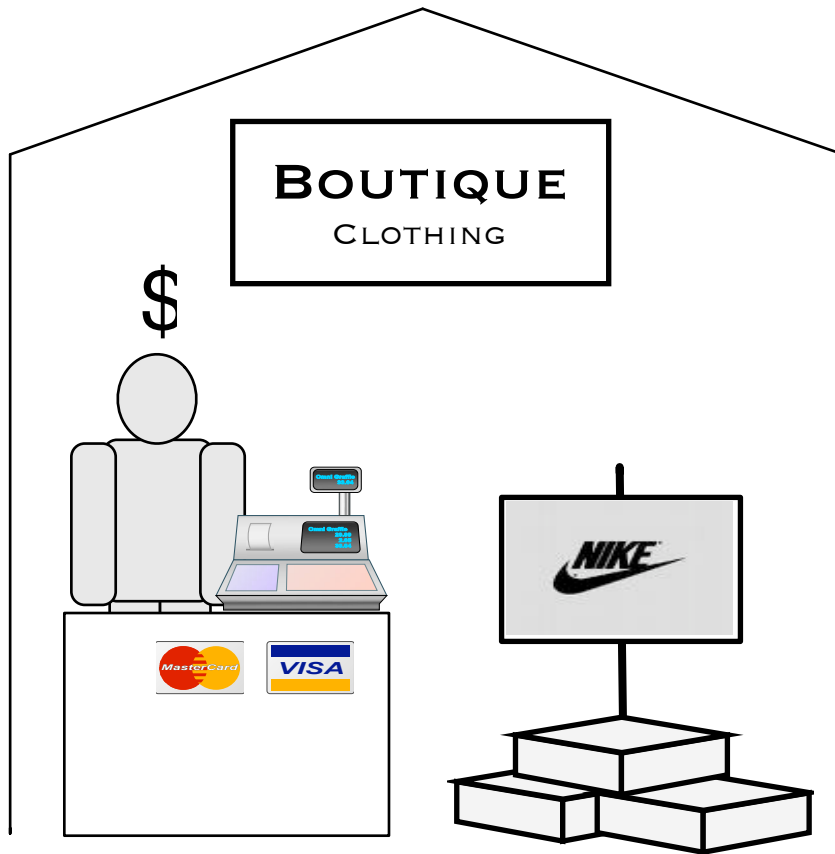
# Running example



# Running example



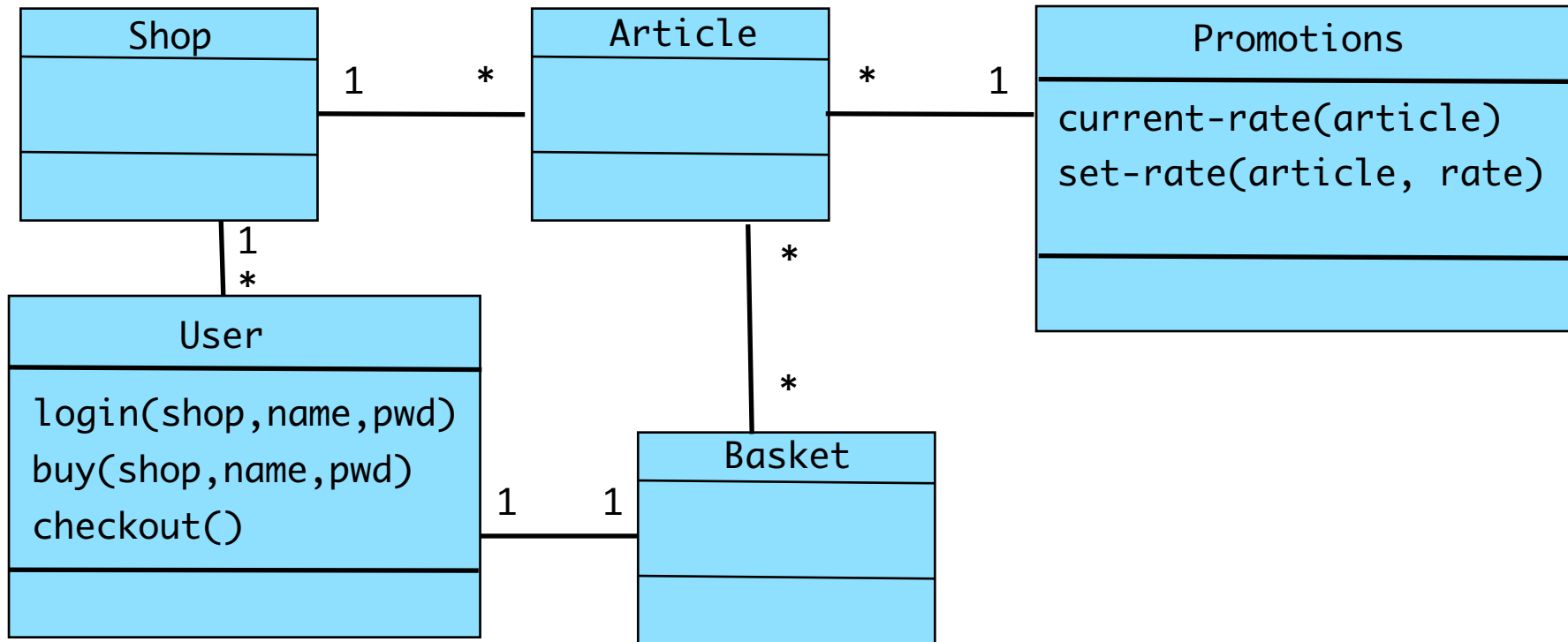
# Running example



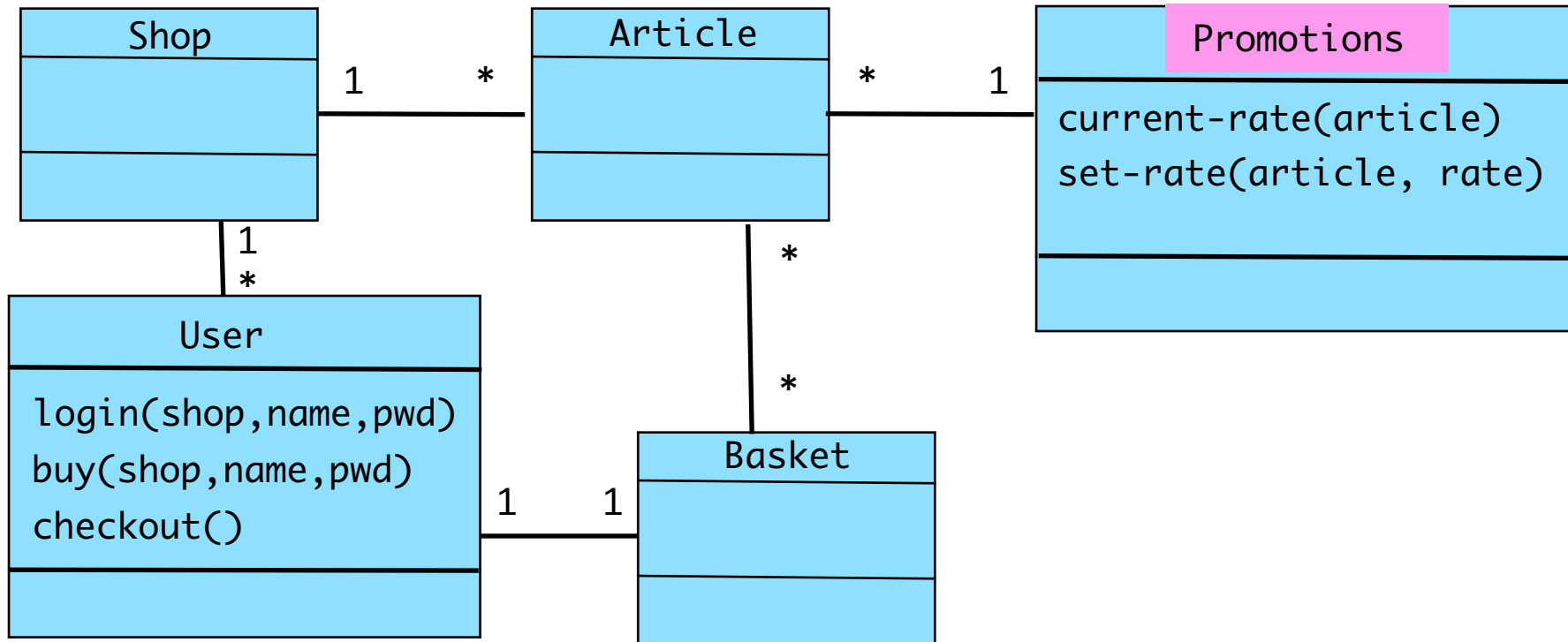


# Running example

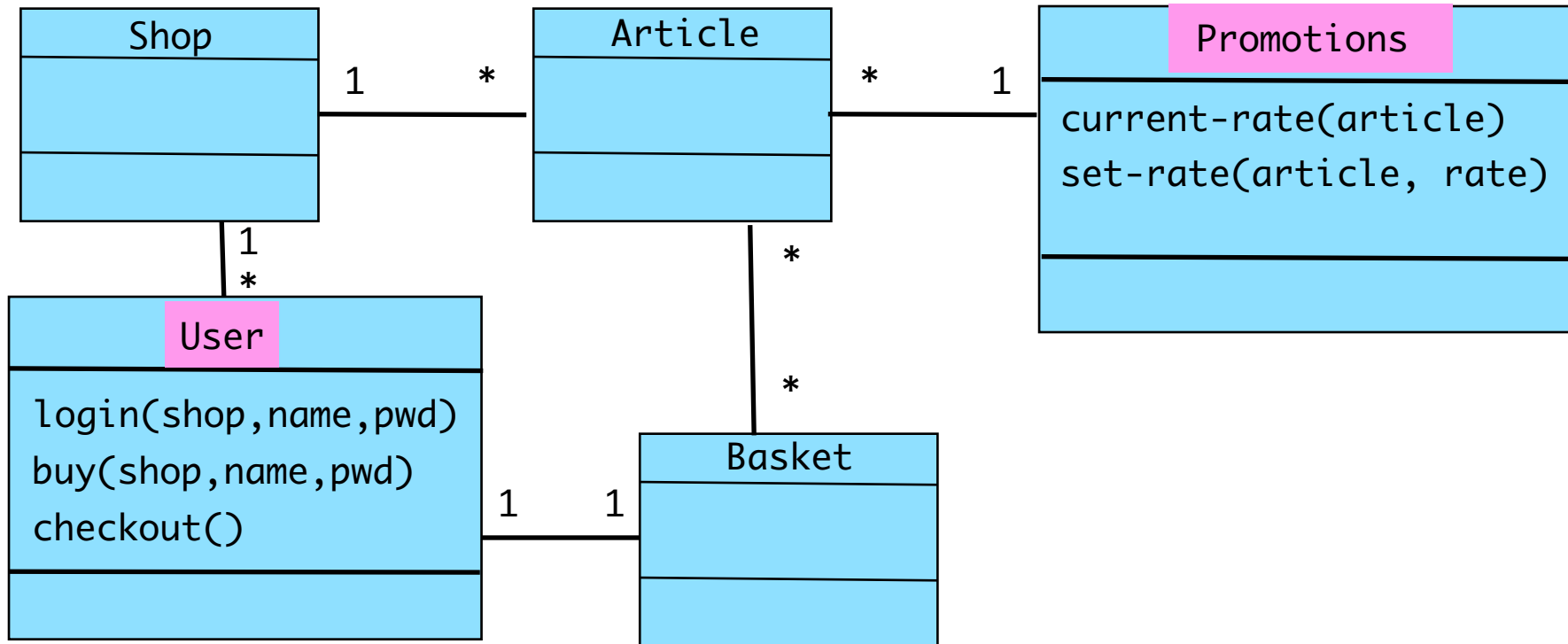
# Running example



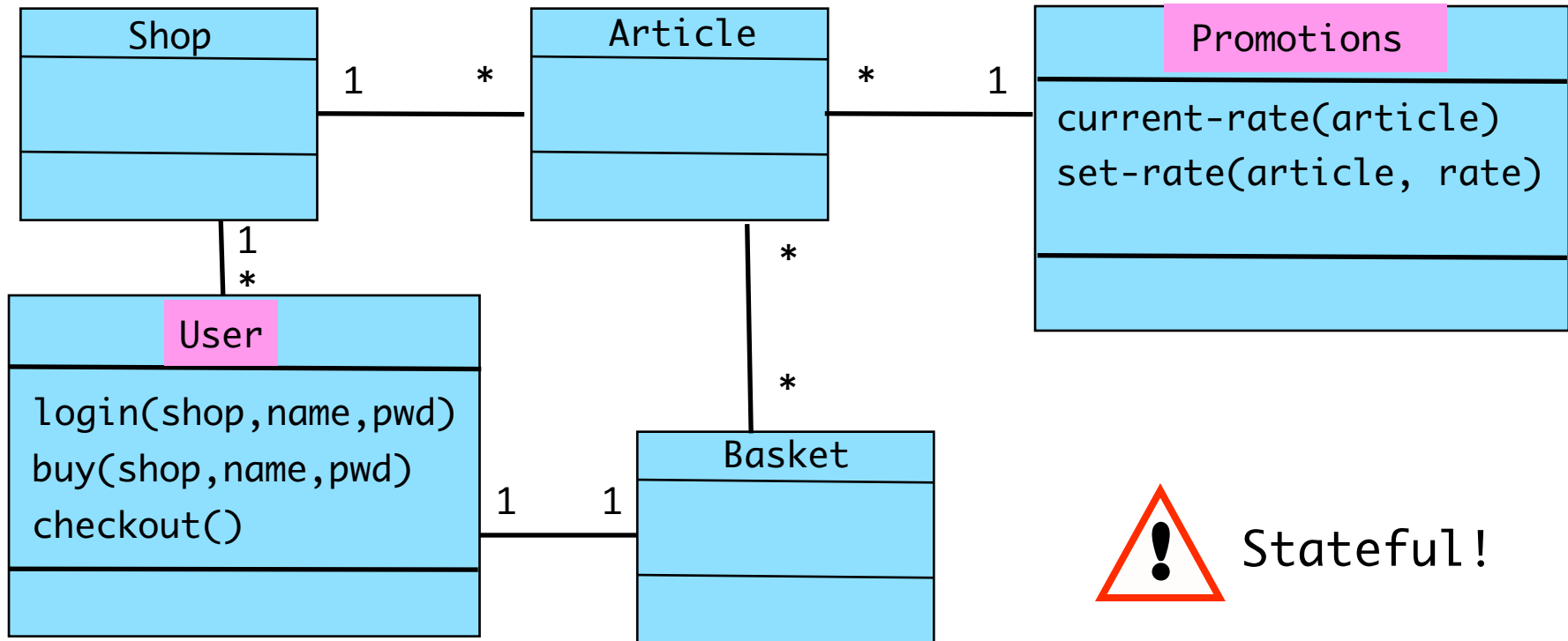
# Running example



# Running example

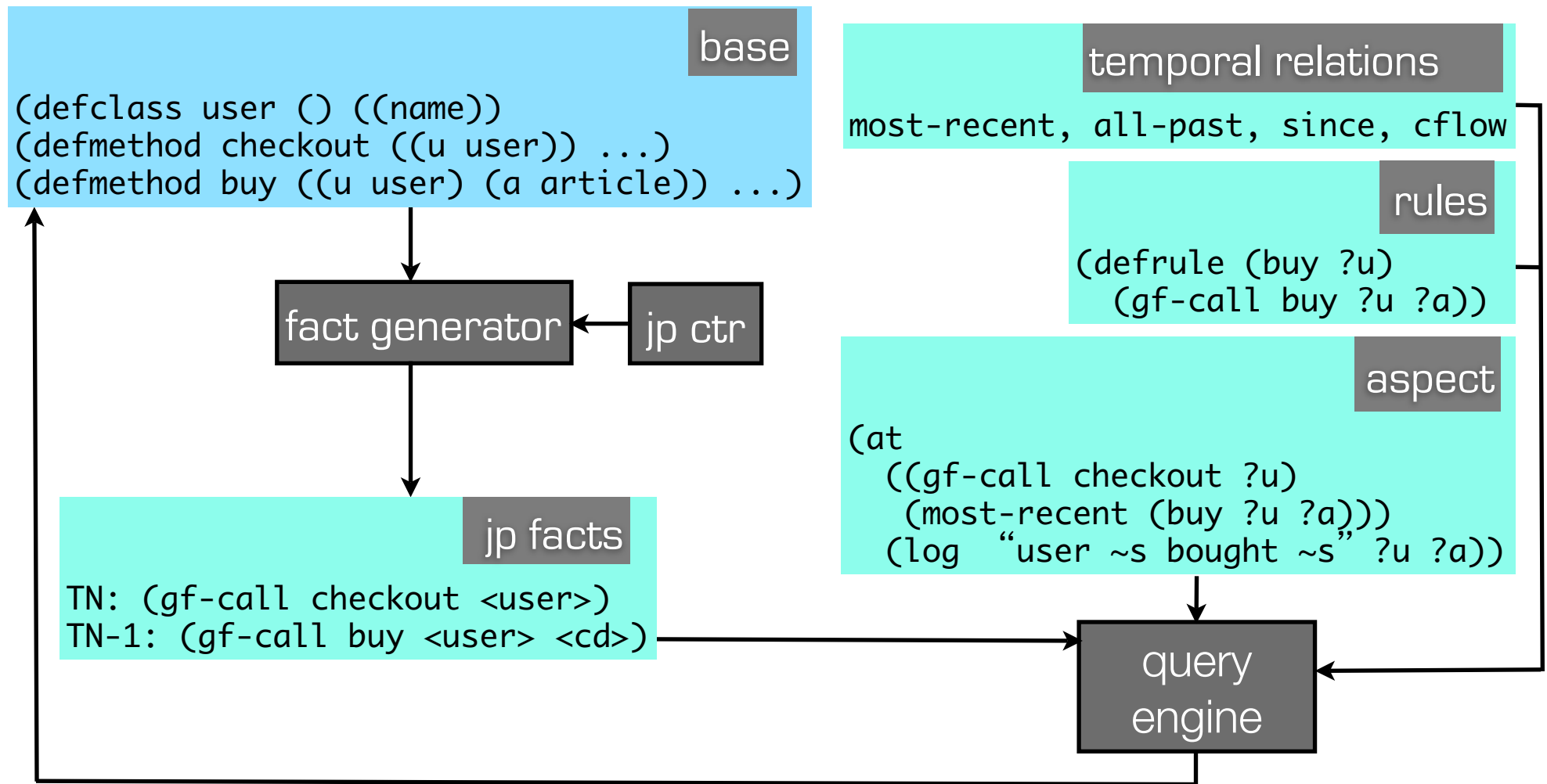


# Running example



Stateful!

# Mental Model



# Join Point Model

## CLOS

```
(make-instance 'user)
```

```
(get-value <kris> 'name)
```

```
(setf (get-value <kris> 'name)  
      "kris")
```

```
(checkout <kris>)
```

## Join Point Facts

```
(create 'user <kris>)
```

```
(slot-get <kris> 'name "kris")
```

```
(slot-set <kris> 'name "" "kris")
```

```
(gf-call checkout <kris>)
```

```
(end-gf-call checkout <kris> nil)
```

# Join Point Model

## CLOS

```
(make-instance 'user)
```

```
(get-value <kris> 'name)
```

```
(setf (get-value <kris> 'name)  
      "kris")
```

```
(checkout <kris>)
```

## Join Point Type Predicates

```
(create 'user ?obj )
```

```
(slot-get ?obj ?name ?val )
```

```
(slot-set ?obj ?name 'oval ?val )
```

```
(gf-call ?gf-name ?obj )
```

```
(end-gf-call ?gf-name ?obj ?res )
```



# Join Point Model

## CLOS

```
(make-instance 'user)
```

```
(get-value <kris> 'name)
```

```
(setf (get-value <kris> 'name)  
      "kris")
```

```
(checkout <kris>)
```

## Join Point Type Predicates

```
(create 'user ?obj )
```

```
(slot-get ?obj ?name ?val )
```

```
(slot-set ?obj ?name 'oval ?val )
```

```
(gf-call ?gf-name ?obj )
```

```
(end-gf-call ?gf-name ?obj ?res )
```

Extensible ...

```
(defrule (promotion "25-12-2008"))
```

# Join Point Model

## CLOS

```
(make-instance 'user)
```

```
(get-value <kris> 'name)
```

```
(setf (get-value <kris> 'name)  
      "kris")
```

```
(checkout <kris>)
```

## Join Point Type Predicates

```
(create 'user ?obj )
```

```
(slot-get ?obj ?name ?val )
```

```
(slot-set ?obj ?name 'oval ?val )
```

```
(gf-call ?gf-name ?obj )
```

```
(end-gf-call ?gf-name ?obj ?res )
```

Extensible ...

```
(defrule [promotion "25-12-2008"])
```

## Hybrid Pointcuts

```
(escape ?name (password ?user))
```

```
(defrule [date ?date] (escape ?date [get-current-time]))
```

# Join Point Model

## CLOS

```
(make-instance 'user)
```

```
(get-value <kris> 'name)
```

```
(setf (get-value <kris> 'name)  
      "kris")
```

```
(checkout <kris>)
```

## Join Point Type Predicates

```
(create 'user ?obj )
```

```
(slot-get ?obj ?name ?val )
```

```
(slot-set ?obj ?name 'oval ?val )
```

```
(gf-call ?gf-name ?obj )
```

```
(end-gf-call ?gf-name ?obj ?res
```

Extensible ...

```
(defrule [promotion "25-12-2008"])
```

## Hybrid Pointcuts

```
(escape ?name (password ?user))
```

```
(defrule [date ?date] (escape ?date [get-current-time]))
```

# Join Point Model

## CLOS

```
(make-instance 'user)
```

```
(get-value <kris> 'name)
```

```
(setf (get-value <kris> 'name)  
      "kris")
```

```
(checkout <kris>)
```



## Join Point Type Predicates

```
(create 'user ?obj )
```

```
(slot-get ?obj ?name ?val )
```

```
(slot-set ?obj ?name 'oval ?val )
```

```
(gf-call ?gf-name ?obj )
```

```
(end-gf-call ?gf-name ?obj ?res )
```

Extensible ...

```
(defrule [promotion "25-12-2008"])
```

## Hybrid Pointcuts

```
(escape ?name (password ?user))
```

```
(defrule [date ?date] [escape ?date [get-current-time]])
```

# Temporal Matching

```
(login <shop> <kris> "kris" "kros" )  
(buy <kris> <jacket>)  
(logout <kris>)  
(login <shop> <kris> "kris" "kros" )  
(buy <kris> <t-shirt>)  
(buy <kris> <socks>)  
(checkout <kris>)
```

# Temporal Matching

```
(login <shop> <kris> "kris" "kros" )  
(buy <kris> <jacket>)  
(logout <kris>)  
(login <shop> <kris> "kris" "kros" )  
(buy <kris> <t-shirt>)  
(buy <kris> <socks>)  
(checkout <kris>)
```



$T_{\text{customer}}$

# Temporal Matching

```
1. (login <shop> <kris> "kris" "kros" )  
   (buy <kris> <jacket>)  
   (logout <kris>)  
   (login <shop> <kris> "kris" "kros" )  
   (buy <kris> <t-shirt>)  
   (buy <kris> <socks>)  
   (checkout <kris>)
```



$T_{\text{customer}}$

# Temporal Matching

```
1. (login <shop> <kris> "kris" "kros" )  
   (buy <kris> <jacket>)  
   (logout <kris>)  
   (login <shop> <kris> "kris" "kros" )  
   (buy <kris> <t-shirt>)  
   (buy <kris> <socks>)  
   (checkout <kris>)
```

login



$T_{\text{customer}}$



# Temporal Matching

```
1. (login <shop> <kris> "kris" "kros" )  
2. (buy <kris> <jacket>)  
   (logout <kris>)  
   (login <shop> <kris> "kris" "kros" )  
   (buy <kris> <t-shirt>)  
   (buy <kris> <socks>)  
   (checkout <kris>)
```

login

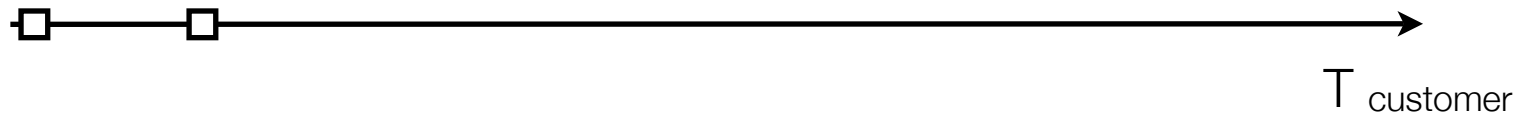


$T_{\text{customer}}$

# Temporal Matching

```
1. (login <shop> <kris> "kris" "kros" )  
2. (buy <kris> <jacket>)  
   (logout <kris>)  
   (login <shop> <kris> "kris" "kros" )  
   (buy <kris> <t-shirt>)  
   (buy <kris> <socks>)  
   (checkout <kris>)
```

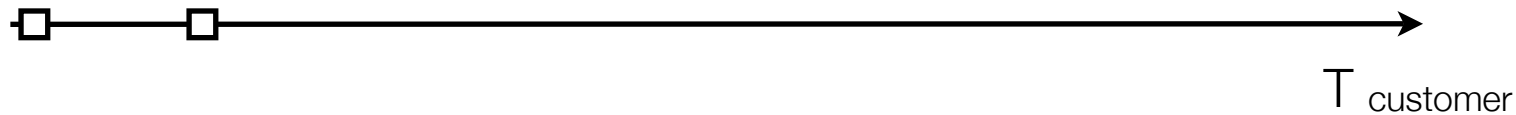
login buy



# Temporal Matching

1. (login <shop> <kris> "kris" "kros" )
  2. (buy <kris> <jacket>)
  3. (logout <kris>)
- (login <shop> <kris> "kris" "kros" )  
(buy <kris> <t-shirt>)  
(buy <kris> <socks>)  
(checkout <kris>)

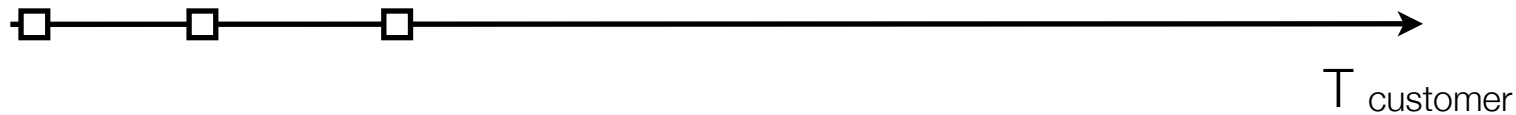
login buy



# Temporal Matching

1. (login <shop> <kris> "kris" "kros" )
  2. (buy <kris> <jacket>)
  3. (logout <kris>)
- (login <shop> <kris> "kris" "kros" )  
(buy <kris> <t-shirt>)  
(buy <kris> <socks>)  
(checkout <kris>)

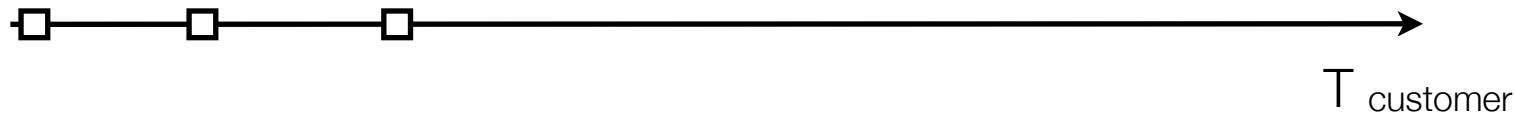
login    buy    logout



# Temporal Matching

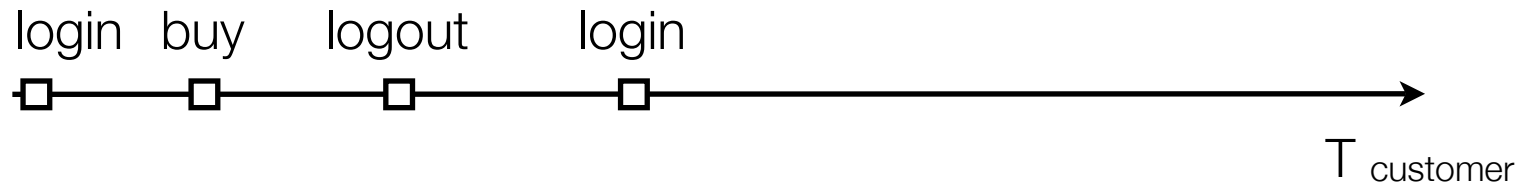
1. (login <shop> <kris> "kris" "kros" )
2. (buy <kris> <jacket>)
3. (logout <kris>)
4. (login <shop> <kris> "kris" "kros" )  
(buy <kris> <t-shirt>)  
(buy <kris> <socks>)  
(checkout <kris>)

login    buy    logout



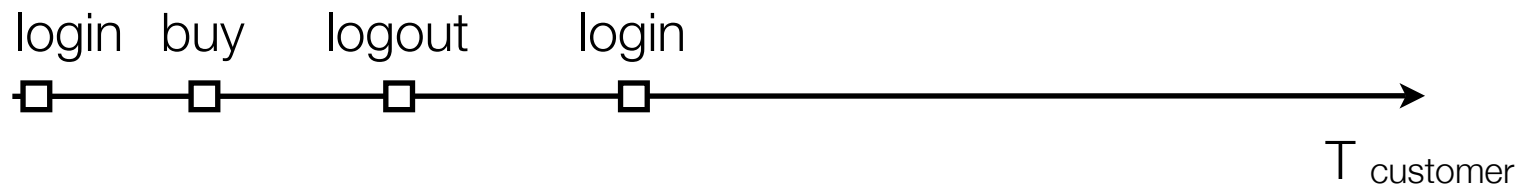
# Temporal Matching

1. (login <shop> <kris> “kris” “kros” )
2. (buy <kris> <jacket>)
3. (logout <kris>)
4. (login <shop> <kris> “kris” “kros” )  
(buy <kris> <t-shirt>)  
(buy <kris> <socks>)  
(checkout <kris>)



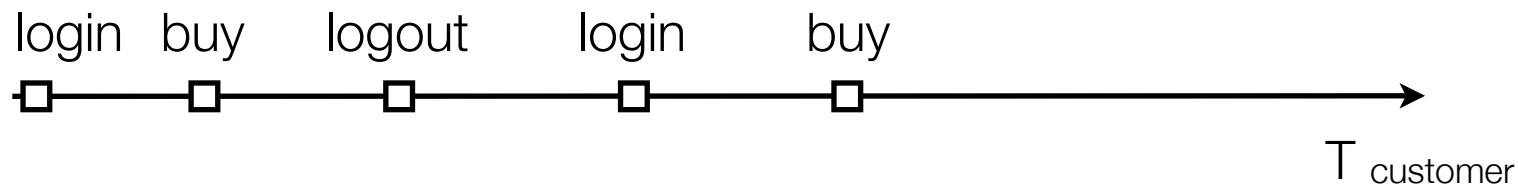
# Temporal Matching

1. (login <shop> <kris> "kris" "kros" )
2. (buy <kris> <jacket>)
3. (logout <kris>)
4. (login <shop> <kris> "kris" "kros" )
5. (buy <kris> <t-shirt>)  
(buy <kris> <socks>)  
(checkout <kris>)



# Temporal Matching

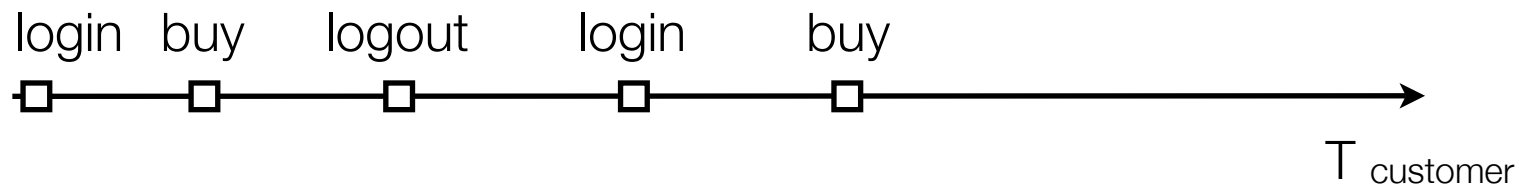
1. (login <shop> <kris> "kris" "kros" )
2. (buy <kris> <jacket>)
3. (logout <kris>)
4. (login <shop> <kris> "kris" "kros" )
5. (buy <kris> <t-shirt>)  
(buy <kris> <socks>)  
(checkout <kris>)





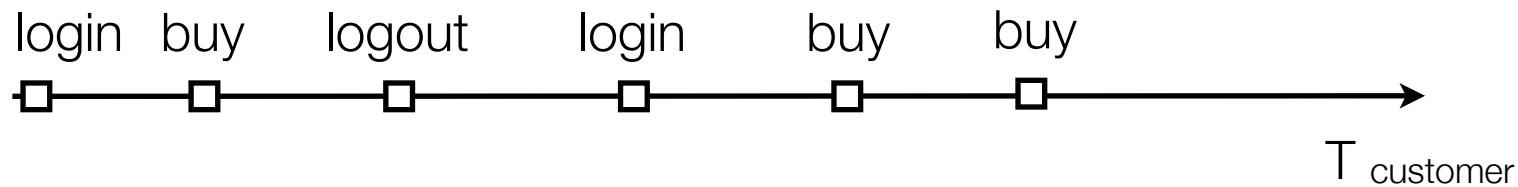
# Temporal Matching

1. (login <shop> <kris> "kris" "kros" )
2. (buy <kris> <jacket>)
3. (logout <kris>)
4. (login <shop> <kris> "kris" "kros" )
5. (buy <kris> <t-shirt>)
6. (buy <kris> <socks>)  
(checkout <kris>)



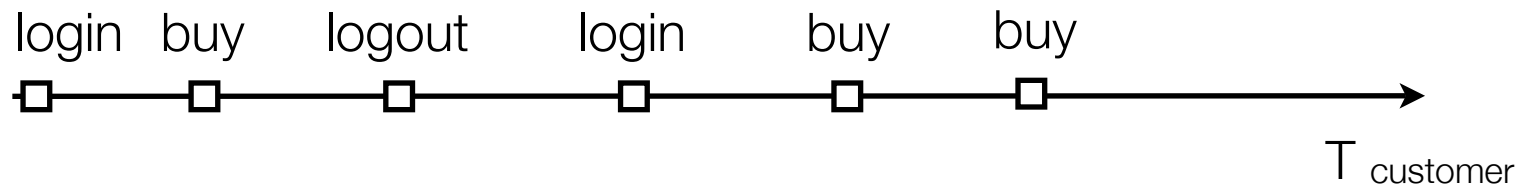
# Temporal Matching

1. (login <shop> <kris> "kris" "kros" )
2. (buy <kris> <jacket>)
3. (logout <kris>)
4. (login <shop> <kris> "kris" "kros" )
5. (buy <kris> <t-shirt>)
6. (buy <kris> <socks>)  
(checkout <kris>)



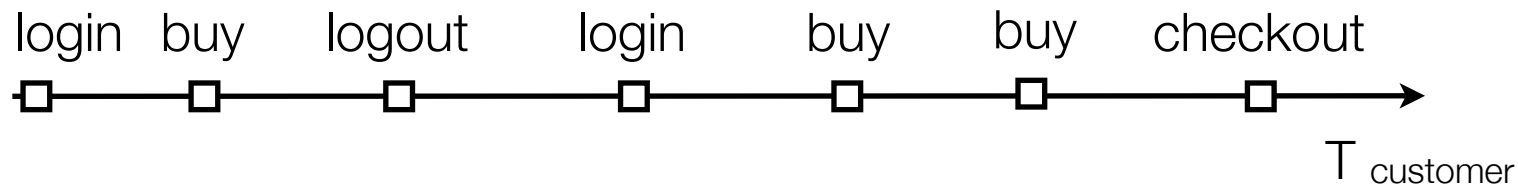
# Temporal Matching

1. (login <shop> <kris> "kris" "kros" )
2. (buy <kris> <jacket>)
3. (logout <kris>)
4. (login <shop> <kris> "kris" "kros" )
5. (buy <kris> <t-shirt>)
6. (buy <kris> <socks>)
7. (checkout <kris>)



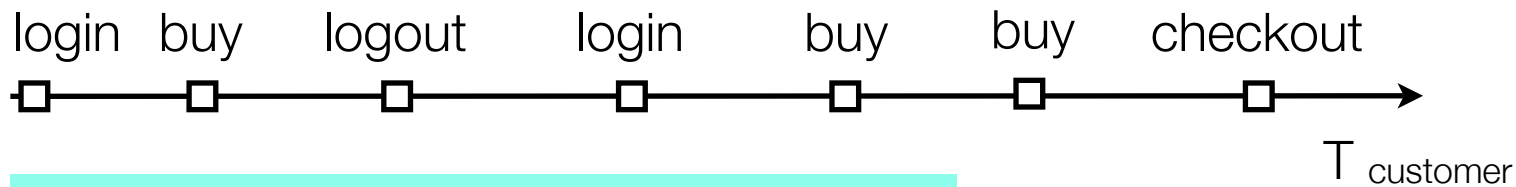
# Temporal Matching

1. (login <shop> <kris> "kris" "kros" )
2. (buy <kris> <jacket>)
3. (logout <kris>)
4. (login <shop> <kris> "kris" "kros" )
5. (buy <kris> <t-shirt>)
6. (buy <kris> <socks>)
7. (checkout <kris>)



# Temporal Matching

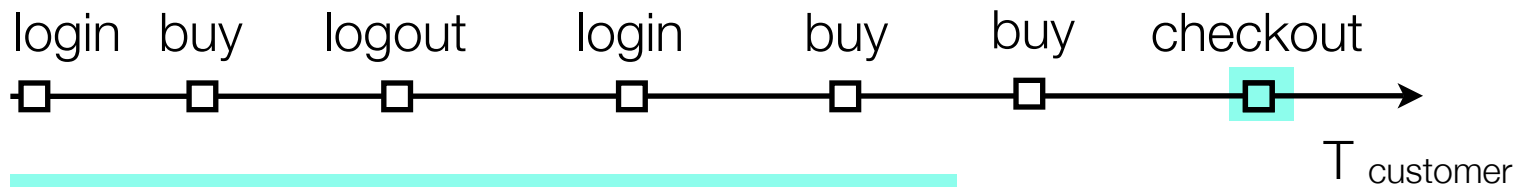
1. (login <shop> <kris> "kris" "kros" )
2. (buy <kris> <jacket>)
3. (logout <kris>)
4. (login <shop> <kris> "kris" "kros" )
5. (buy <kris> <t-shirt>)
6. (buy <kris> <socks>)
7. (checkout <kris>)



```
((gf-call checkout ?args)
  (most-recent (gf-call buy ?args2)))
```

# Temporal Matching

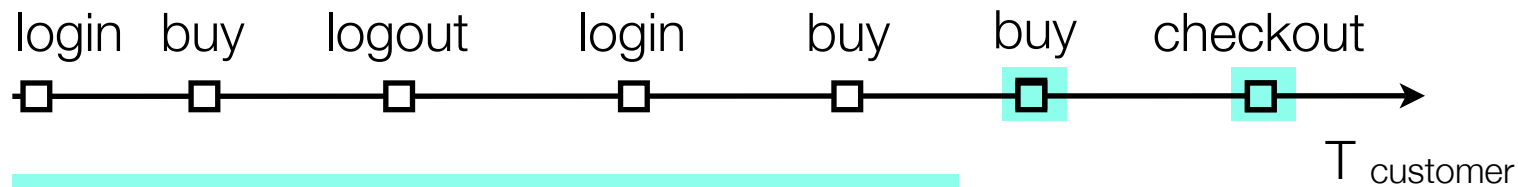
1. (login <shop> <kris> "kris" "kros" )
2. (buy <kris> <jacket>)
3. (logout <kris>)
4. (login <shop> <kris> "kris" "kros" )
5. (buy <kris> <t-shirt>)
6. (buy <kris> <socks>)
7. (checkout <kris>)



```
((gf-call checkout ?args)
  (most-recent (gf-call buy ?args2)))
```

# Temporal Matching

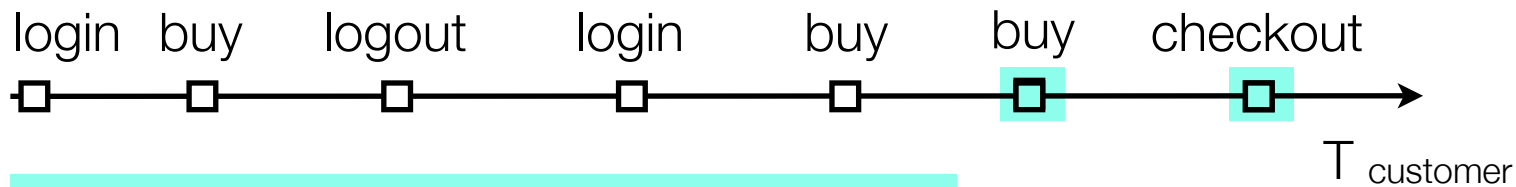
1. (login <shop> <kris> "kris" "kros" )
2. (buy <kris> <jacket>)
3. (logout <kris>)
4. (login <shop> <kris> "kris" "kros" )
5. (buy <kris> <t-shirt>)
6. (buy <kris> <socks>)
7. (checkout <kris>)



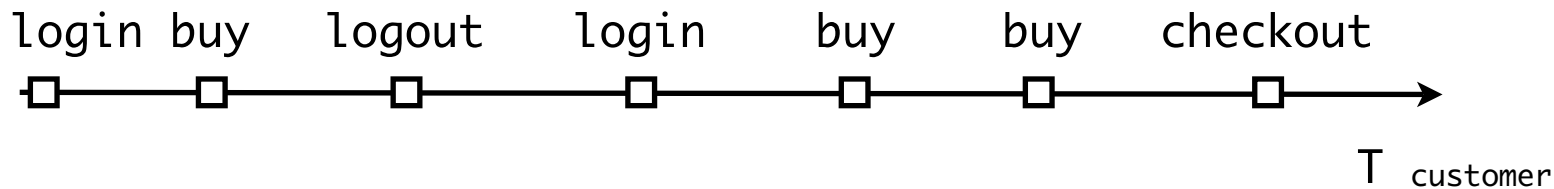
```
((gf-call checkout ?args)
  (most-recent (gf-call buy ?args2)))
```

# Temporal Matching

1. (login <shop> <kris> "kris" "kros" )
2. (buy <kris> <jacket>)
3. (logout <kris>)
4. (login <shop> <kris> "kris" "kros" )
5. (buy <kris> <t-shirt>)
6. (buy <kris> <socks>)
7. (checkout <kris>)



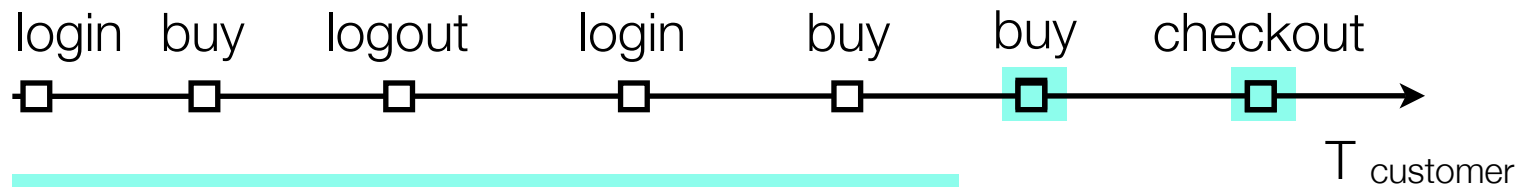
```
((gf-call checkout ?args)
  (most-recent (gf-call buy ?args2)))
```



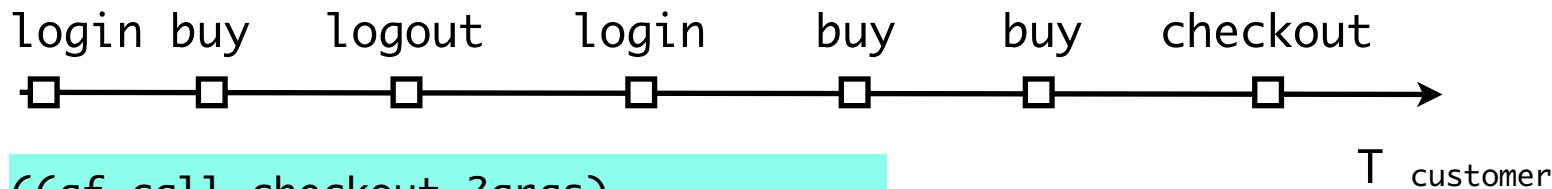


# Temporal Matching

1. (login <shop> <kris> "kris" "kros" )
2. (buy <kris> <jacket>)
3. (logout <kris>)
4. (login <shop> <kris> "kris" "kros" )
5. (buy <kris> <t-shirt>)
6. (buy <kris> <socks>)
7. (checkout <kris>)



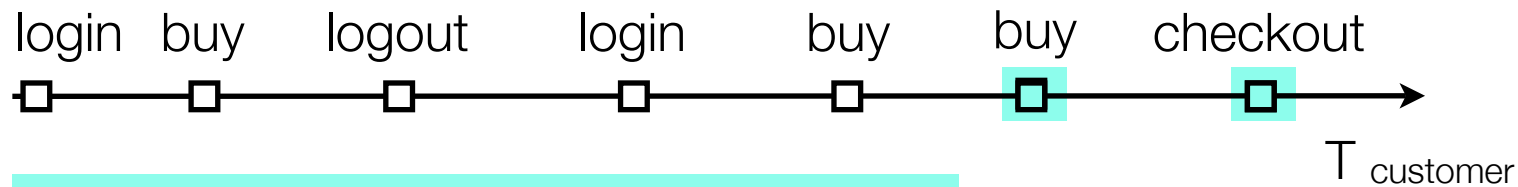
```
((gf-call checkout ?args)
  (most-recent (gf-call buy ?args2)))
```



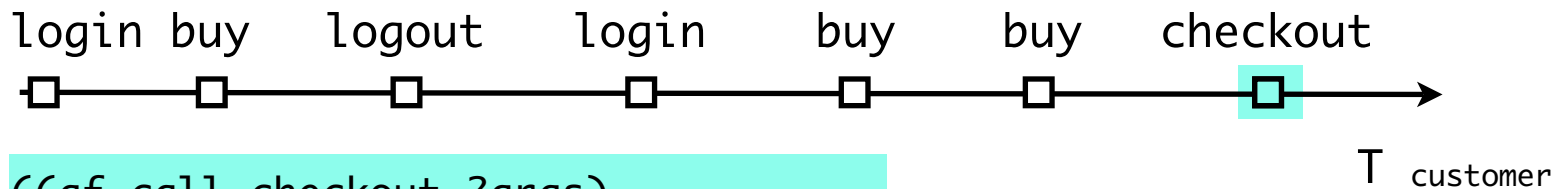
```
((gf-call checkout ?args)
  (all-past (gf-call buy ?args2)))
```

# Temporal Matching

1. (login <shop> <kris> "kris" "kros" )
2. (buy <kris> <jacket>)
3. (logout <kris>)
4. (login <shop> <kris> "kris" "kros" )
5. (buy <kris> <t-shirt>)
6. (buy <kris> <socks>)
7. (checkout <kris>)



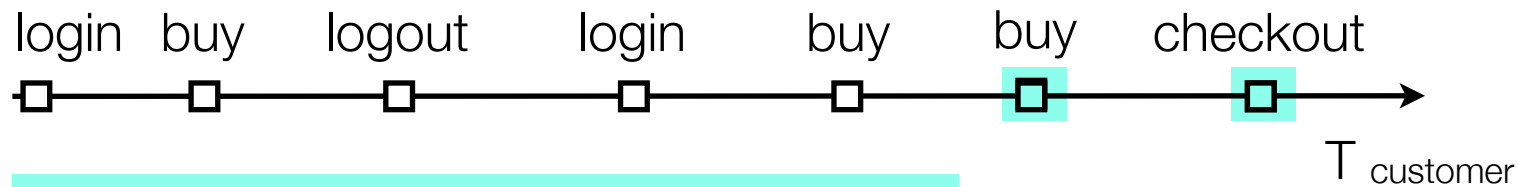
```
((gf-call checkout ?args)
  (most-recent (gf-call buy ?args2)))
```



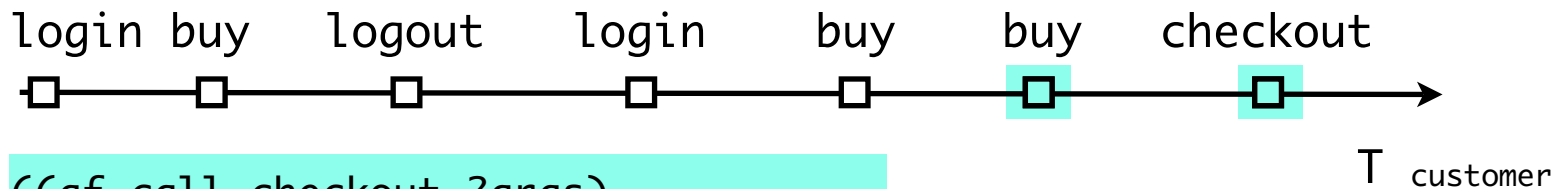
```
((gf-call checkout ?args)
  (all-past (gf-call buy ?args2)))
```

# Temporal Matching

1. (login <shop> <kris> "kris" "kros" )
2. (buy <kris> <jacket>)
3. (logout <kris>)
4. (login <shop> <kris> "kris" "kros" )
5. (buy <kris> <t-shirt>)
6. (buy <kris> <socks>)
7. (checkout <kris>)



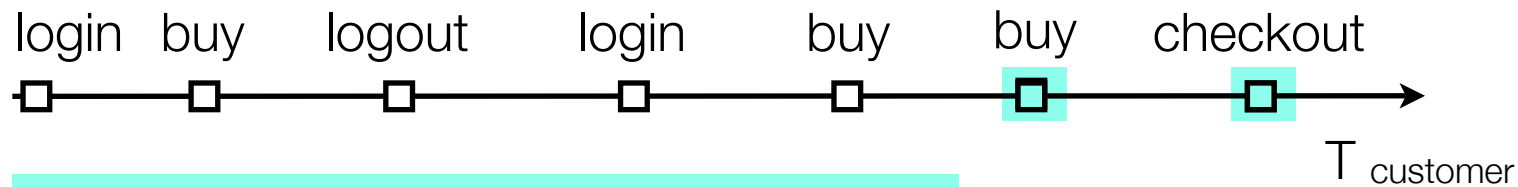
```
((gf-call checkout ?args)
  (most-recent (gf-call buy ?args2)))
```



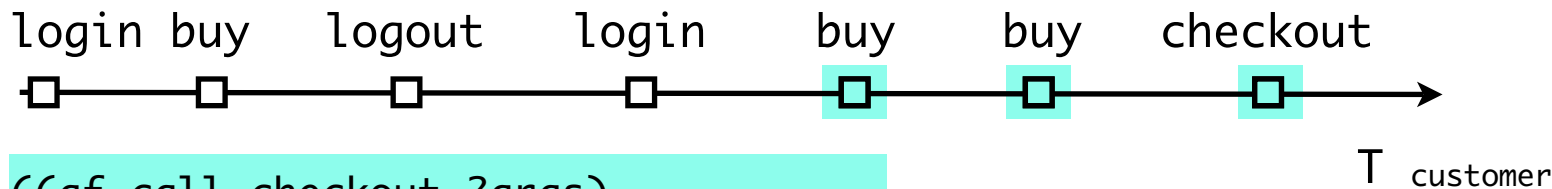
```
((gf-call checkout ?args)
  (all-past (gf-call buy ?args2)))
```

# Temporal Matching

1. (login <shop> <kris> "kris" "kros" )
2. (buy <kris> <jacket>)
3. (logout <kris>)
4. (login <shop> <kris> "kris" "kros" )
5. (buy <kris> <t-shirt>)
6. (buy <kris> <socks>)
7. (checkout <kris>)



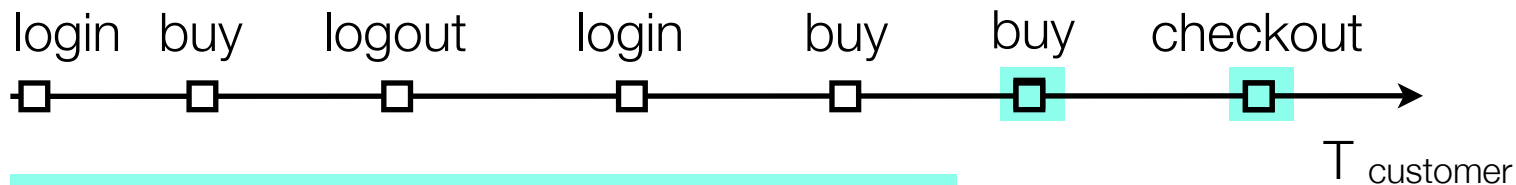
```
((gf-call checkout ?args)
  (most-recent (gf-call buy ?args2)))
```



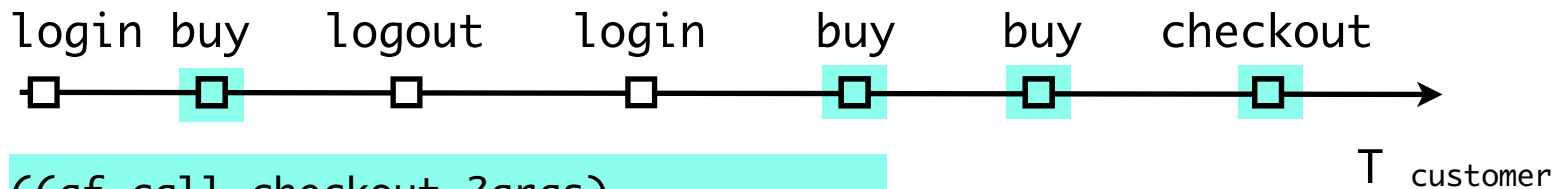
```
((gf-call checkout ?args)
  (all-past (gf-call buy ?args2)))
```

# Temporal Matching

1. (login <shop> <kris> "kris" "kros" )
2. (buy <kris> <jacket>)
3. (logout <kris>)
4. (login <shop> <kris> "kris" "kros" )
5. (buy <kris> <t-shirt>)
6. (buy <kris> <socks>)
7. (checkout <kris>)



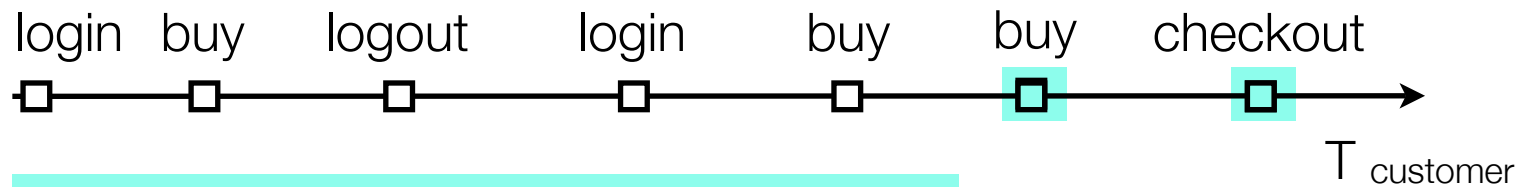
```
((gf-call checkout ?args)
  (most-recent (gf-call buy ?args2)))
```



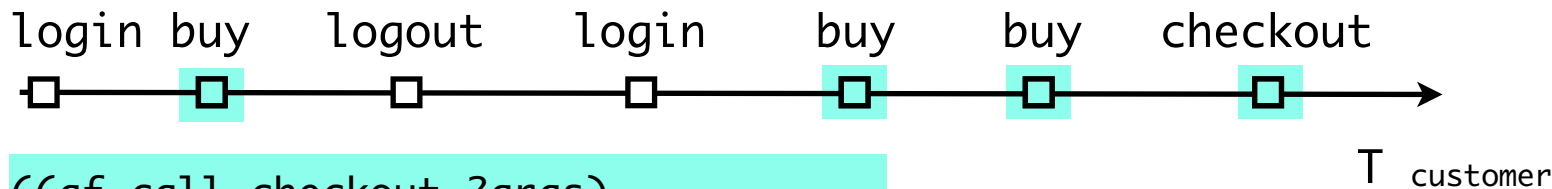
```
((gf-call checkout ?args)
  (all-past (gf-call buy ?args2)))
```

# Temporal Matching

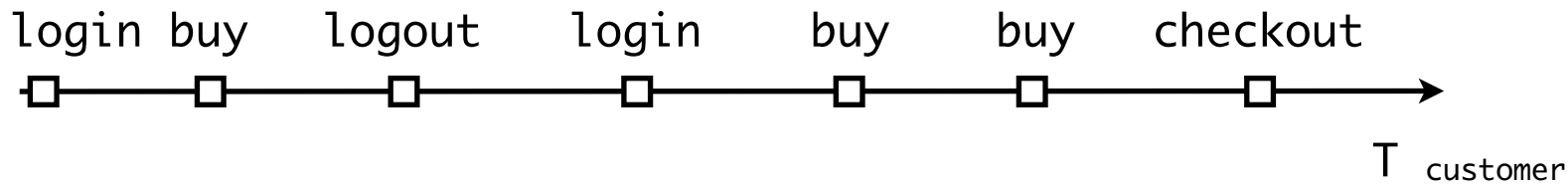
1. (login <shop> <kris> "kris" "kros" )
2. (buy <kris> <jacket>)
3. (logout <kris>)
4. (login <shop> <kris> "kris" "kros" )
5. (buy <kris> <t-shirt>)
6. (buy <kris> <socks>)
7. (checkout <kris>)



```
((gf-call checkout ?args)
  (most-recent (gf-call buy ?args2)))
```

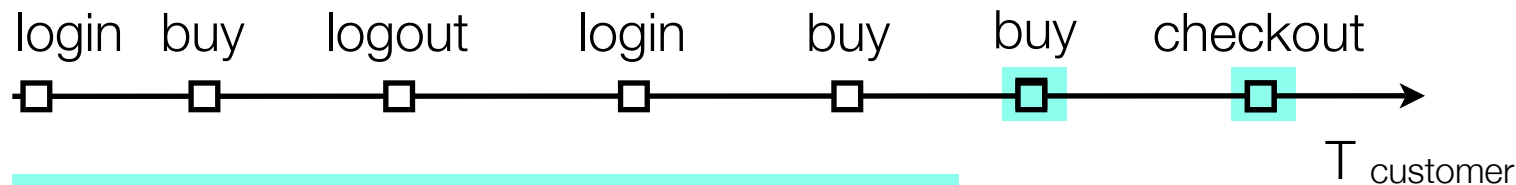


```
((gf-call checkout ?args)
  (all-past (gf-call buy ?args2)))
```

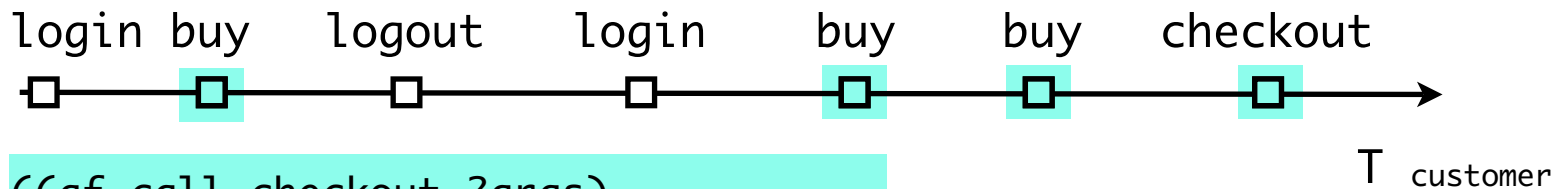


# Temporal Matching

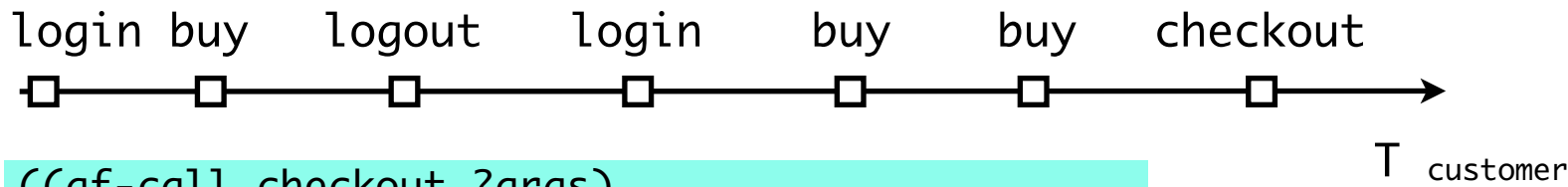
1. (login <shop> <kris> "kris" "kros" )
2. (buy <kris> <jacket>)
3. (logout <kris>)
4. (login <shop> <kris> "kris" "kros" )
5. (buy <kris> <t-shirt>)
6. (buy <kris> <socks>)
7. (checkout <kris>)



```
((gf-call checkout ?args)
  (most-recent (gf-call buy ?args2)))
```



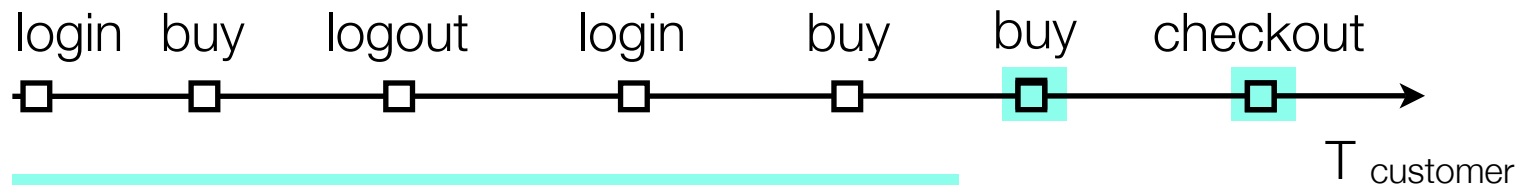
```
((gf-call checkout ?args)
  (all-past (gf-call buy ?args2)))
```



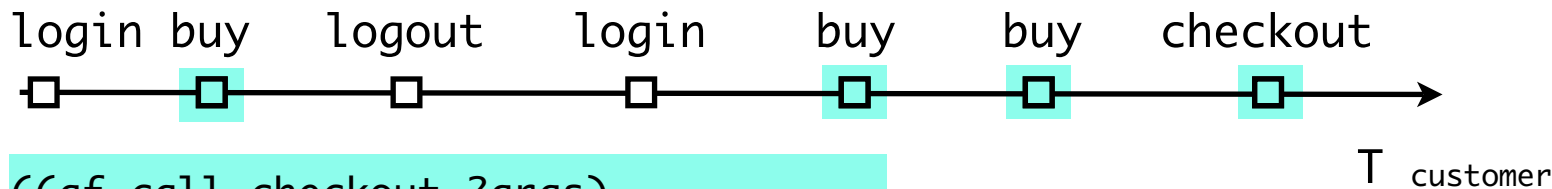
```
((gf-call checkout ?args)
  (since (most-recent (gf-call login ?args2))
    (all-past (gf-call buy ?args3))))
```

# Temporal Matching

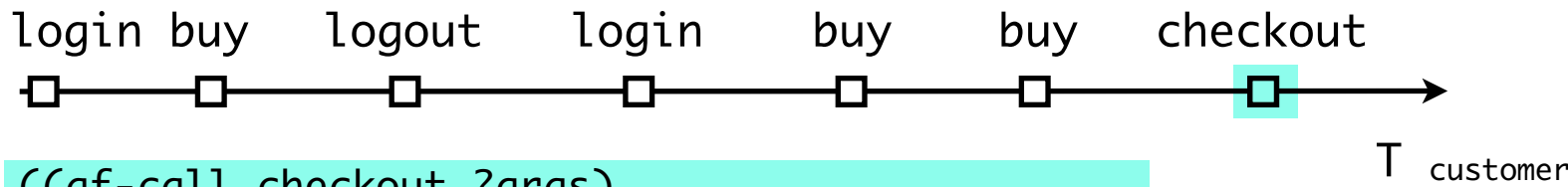
1. (login <shop> <kris> "kris" "kros" )
2. (buy <kris> <jacket>)
3. (logout <kris>)
4. (login <shop> <kris> "kris" "kros" )
5. (buy <kris> <t-shirt>)
6. (buy <kris> <socks>)
7. (checkout <kris>)



```
((gf-call checkout ?args)
  (most-recent (gf-call buy ?args2)))
```



```
((gf-call checkout ?args)
  (all-past (gf-call buy ?args2)))
```

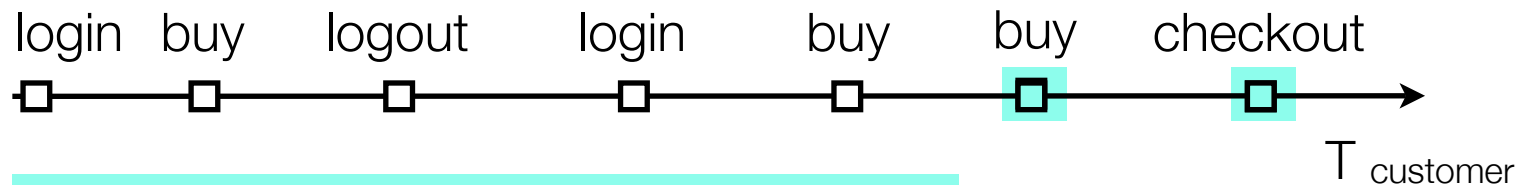


```
((gf-call checkout ?args)
  (since (most-recent (gf-call login ?args2))
    (all-past (gf-call buy ?args3))))
```

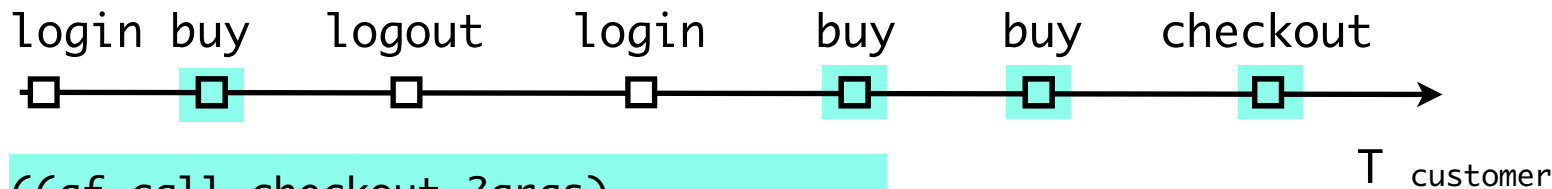


# Temporal Matching

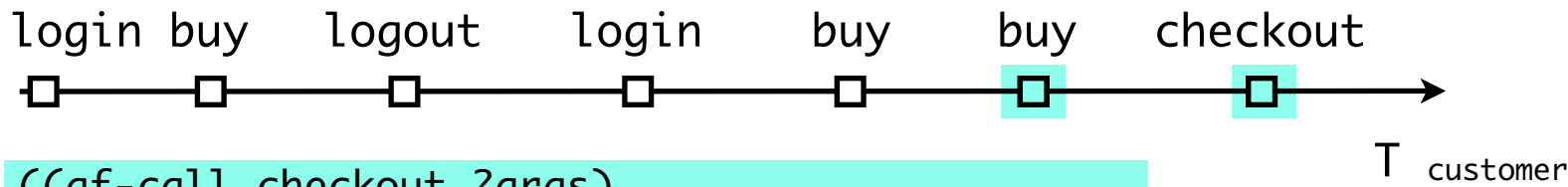
1. (login <shop> <kris> "kris" "kros" )
2. (buy <kris> <jacket>)
3. (logout <kris>)
4. (login <shop> <kris> "kris" "kros" )
5. (buy <kris> <t-shirt>)
6. (buy <kris> <socks>)
7. (checkout <kris>)



```
((gf-call checkout ?args)
  (most-recent (gf-call buy ?args2)))
```



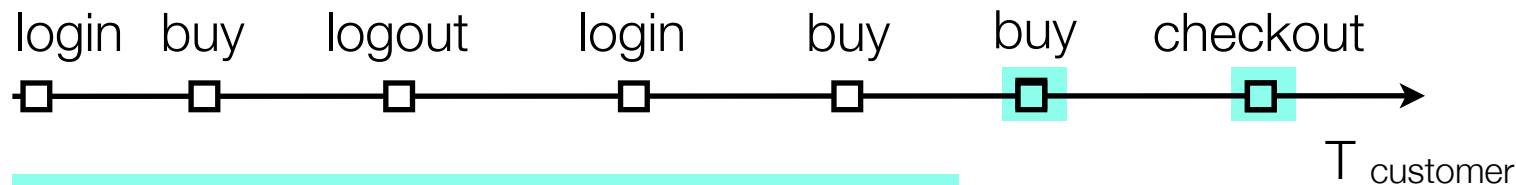
```
((gf-call checkout ?args)
  (all-past (gf-call buy ?args2)))
```



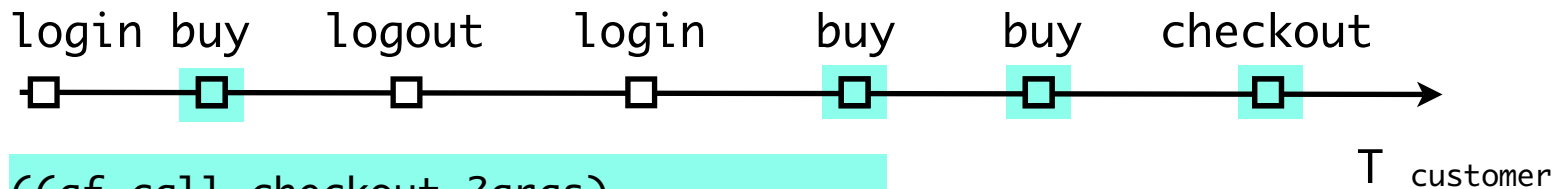
```
((gf-call checkout ?args)
  (since (most-recent (gf-call login ?args2))
    (all-past (gf-call buy ?args3))))
```

# Temporal Matching

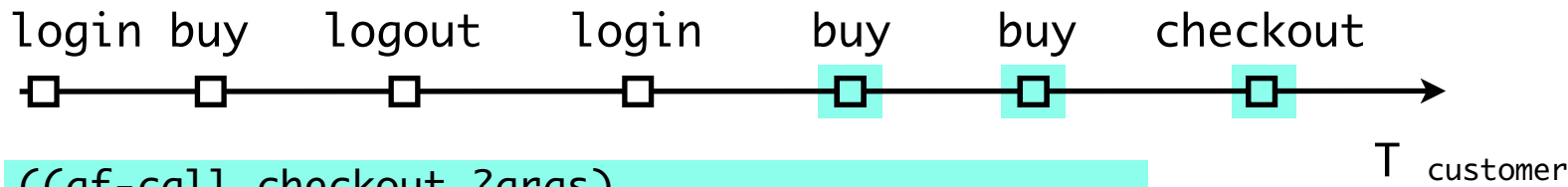
1. (login <shop> <kris> "kris" "kros" )
2. (buy <kris> <jacket>)
3. (logout <kris>)
4. (login <shop> <kris> "kris" "kros" )
5. (buy <kris> <t-shirt>)
6. (buy <kris> <socks>)
7. (checkout <kris>)



```
((gf-call checkout ?args)
  (most-recent (gf-call buy ?args2)))
```



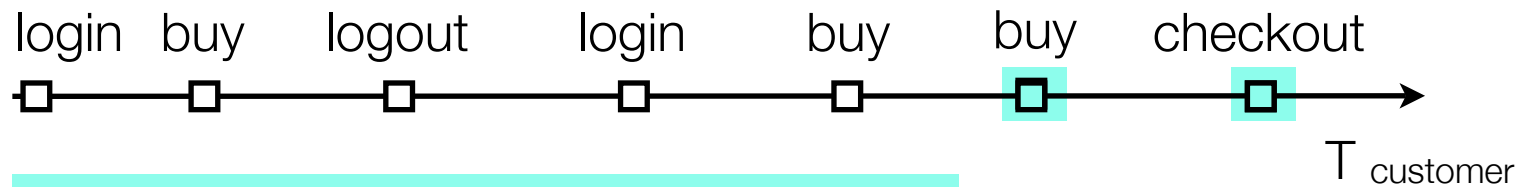
```
((gf-call checkout ?args)
  (all-past (gf-call buy ?args2)))
```



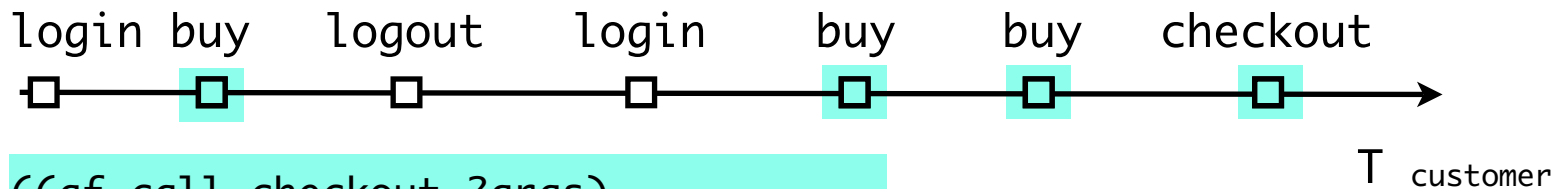
```
((gf-call checkout ?args)
  (since (most-recent (gf-call login ?args2))
    (all-past (gf-call buy ?args3))))
```

# Temporal Matching

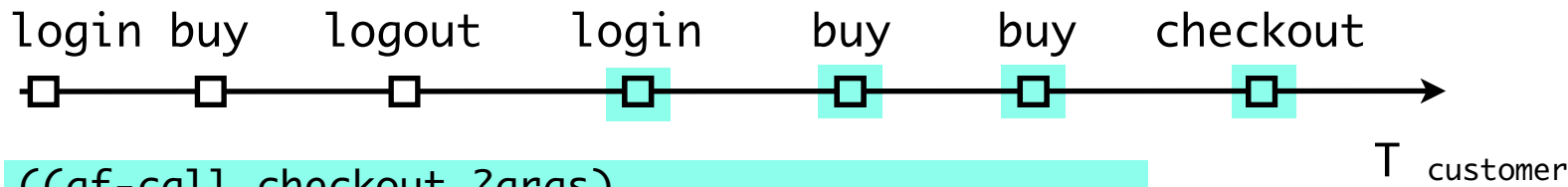
1. (login <shop> <kris> "kris" "kros" )
2. (buy <kris> <jacket>)
3. (logout <kris>)
4. (login <shop> <kris> "kris" "kros" )
5. (buy <kris> <t-shirt>)
6. (buy <kris> <socks>)
7. (checkout <kris>)



```
((gf-call checkout ?args)
  (most-recent (gf-call buy ?args2)))
```



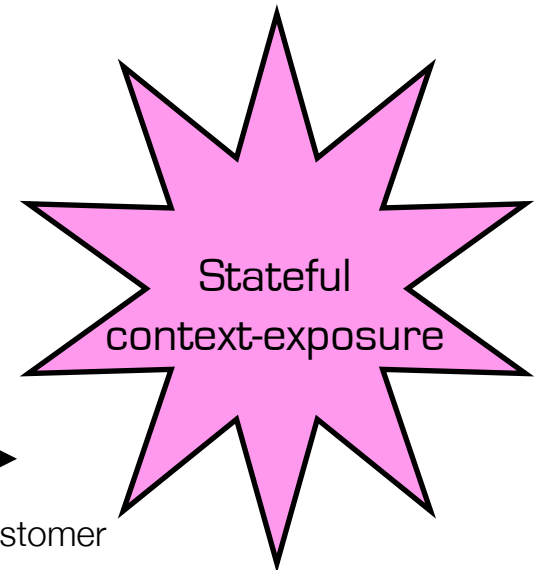
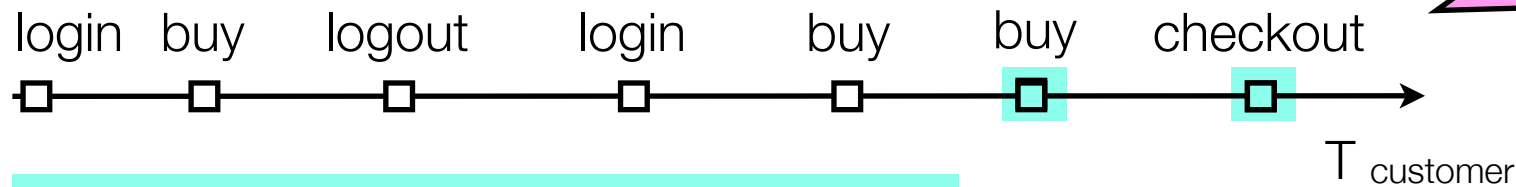
```
((gf-call checkout ?args)
  (all-past (gf-call buy ?args2)))
```



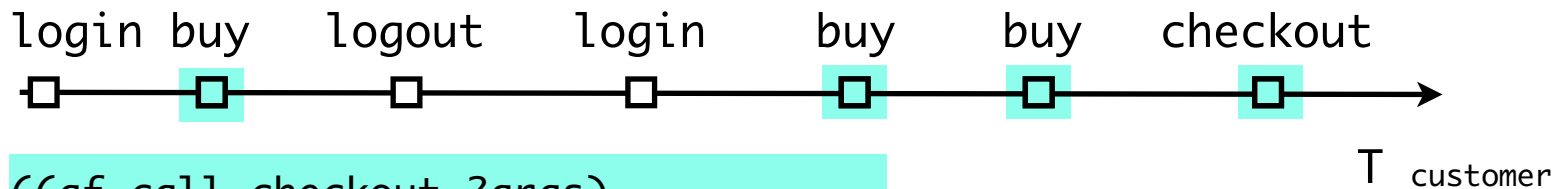
```
((gf-call checkout ?args)
  (since (most-recent (gf-call login ?args2))
    (all-past (gf-call buy ?args3))))
```

# Temporal Matching

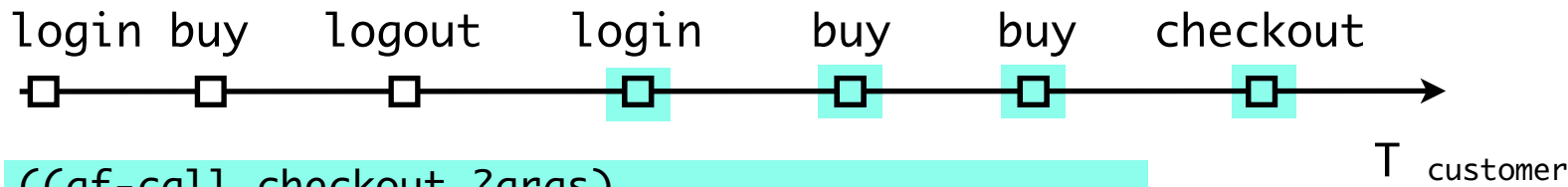
1. (login <shop> <kris> "kris" "kros" )
2. (buy <kris> <jacket>)
3. (logout <kris>)
4. (login <shop> <kris> "kris" "kros" )
5. (buy <kris> <t-shirt>)
6. (buy <kris> <socks>)
7. (checkout <kris>)



```
((gf-call checkout ?args)
  (most-recent (gf-call buy ?args2)))
```



```
((gf-call checkout ?args)
  (all-past (gf-call buy ?args2)))
```



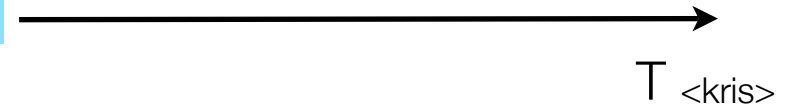
```
((gf-call checkout ?args)
  (since (most-recent (gf-call login ?args2))
    (all-past (gf-call buy ?args3))))
```

# Stateful context-exposure

```
(let ((promo (singleton-instance 'promotions)))  
  (setf (rate-for promo) <cd> 0.05)  
  (login <kris> <shop> "kris" "kros" )  
  (buy <kris> <cd>)  
  (setf (rate-for promo) <cd> 0.00)  
  (buy <kris> <cd>)))
```

# Stateful context-exposure

```
(let ((promo (singleton-instance 'promotions)))  
  (setf (rate-for promo) <cd> 0.05)  
  (login <kris> <shop> "kris" "kros" )  
  (buy <kris> <cd>)  
  (setf (rate-for promo) <cd> 0.00)  
  (buy <kris> <cd>)))
```



# Stateful context-exposure

```
(let ((promo (singleton-instance 'promotions)))  
  (setf (rate-for promo) <cd> 0.05)  
  (login <kris> <shop> "kris" "kros" )  
  (buy <kris> <cd>)  
  (setf (rate-for promo) <cd> 0.00)  
  (buy <kris> <cd>)))
```

T <kris>

```
(at  
  ((gf-call buy ?user ?article)  
   (most-recent (gf-call login ?user _ _ _)  
                 (rate-for ?rate ?article)))  
  (format t "~s gets a ~s % discount on ~s" ?user ?rate ?article))
```

# Stateful context-exposure

```
1.(let ((promo (singleton-instance 'promotions)))  
  (setf (rate-for promo) <cd> 0.05)  
  (login <kris> <shop> "kris" "kros" )  
  (buy <kris> <cd>)  
  (setf (rate-for promo) <cd> 0.00)  
  (buy <kris> <cd>)))
```

T <kris>

```
(at  
  ((gf-call buy ?user ?article)  
   (most-recent (gf-call login ?user _ _ _)  
                 (rate-for ?rate ?article)))  
  (format t "~s gets a ~s % discount on ~s" ?user ?rate ?article))
```



# Stateful context-exposure

```
1. (let ((promo (singleton-instance 'promotions)))  
2.  (setf (rate-for promo) <cd> 0.05)  
    (login <kris> <shop> "kris" "kros" )  
    (buy <kris> <cd>)  
    (setf (rate-for promo) <cd> 0.00)  
    (buy <kris> <cd>)))
```

T <kris>

```
(at  
  ((gf-call buy ?user ?article)  
   (most-recent (gf-call login ?user _ _ _)  
                 (rate-for ?rate ?article)))  
  (format t "~s gets a ~s % discount on ~s" ?user ?rate ?article))
```

# Stateful context-exposure

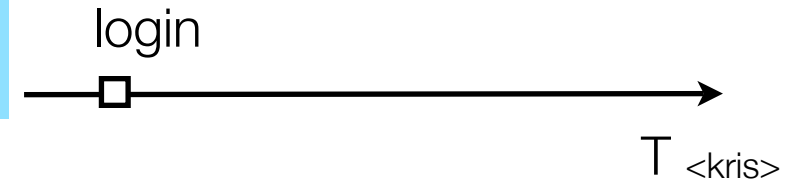
```
1. (let ((promo (singleton-instance 'promotions)))  
2.   (setf (rate-for promo) <cd> 0.05)  
3.   (login <kris> <shop> "kris" "kros" )  
    (buy <kris> <cd>)  
    (setf (rate-for promo) <cd> 0.00)  
    (buy <kris> <cd>)))
```

T <kris>

```
(at  
  ((gf-call buy ?user ?article)  
   (most-recent (gf-call login ?user _ _ _)  
                 (rate-for ?rate ?article)))  
  (format t "~s gets a ~s % discount on ~s" ?user ?rate ?article))
```

# Stateful context-exposure

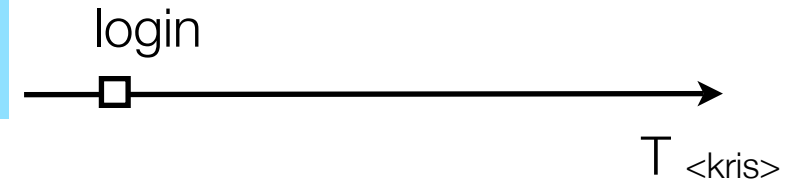
```
1. (let ((promo (singleton-instance 'promotions)))  
2.   (setf (rate-for promo) <cd> 0.05)  
3.   (login <kris> <shop> "kris" "kros" )  
    (buy <kris> <cd>)  
    (setf (rate-for promo) <cd> 0.00)  
    (buy <kris> <cd>)))
```



```
(at  
  ((gf-call buy ?user ?article)  
   (most-recent (gf-call login ?user _ _ _)  
                 (rate-for ?rate ?article)))  
  (format t "~s gets a ~s % discount on ~s" ?user ?rate ?article))
```

# Stateful context-exposure

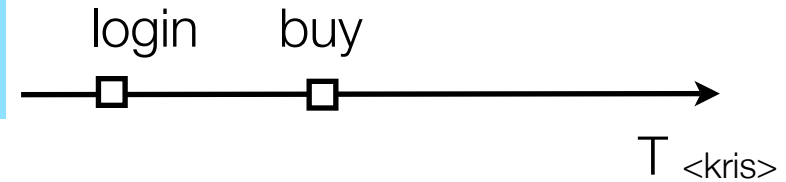
```
1. (let ((promo (singleton-instance 'promotions)))
2.   (setf (rate-for promo) <cd> 0.05)
3.   (login <kris> <shop> "kris" "kros" )
4.   (buy <kris> <cd>)
   (setf (rate-for promo) <cd> 0.00)
   (buy <kris> <cd>)))
```



```
(at
  ((gf-call buy ?user ?article)
   (most-recent (gf-call login ?user _ _ _)
                 (rate-for ?rate ?article)))
  (format t "~s gets a ~s % discount on ~s" ?user ?rate ?article))
```

# Stateful context-exposure

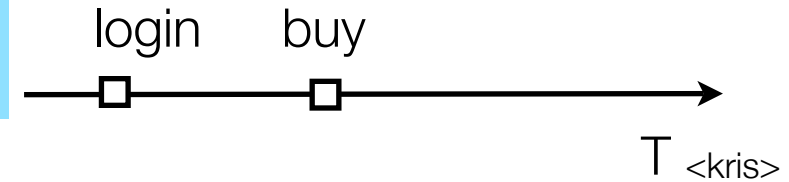
```
1. (let ((promo (singleton-instance 'promotions)))  
2.   (setf (rate-for promo) <cd> 0.05)  
3.   (login <kris> <shop> "kris" "kros" )  
4.   (buy <kris> <cd>)  
    (setf (rate-for promo) <cd> 0.00)  
    (buy <kris> <cd>)))
```



```
(at  
  ((gf-call buy ?user ?article)  
   (most-recent (gf-call login ?user _ _ _)  
                 (rate-for ?rate ?article)))  
  (format t "~s gets a ~s % discount on ~s" ?user ?rate ?article))
```

# Stateful context-exposure

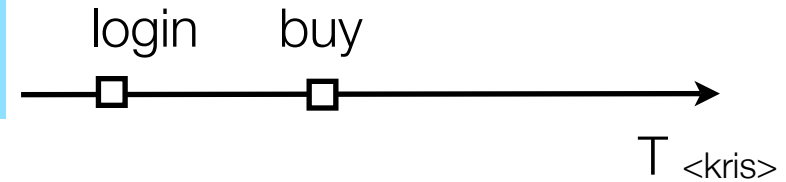
```
1. (let ((promo (singleton-instance 'promotions)))  
2.   (setf (rate-for promo) <cd> 0.05)  
3.   (login <kris> <shop> "kris" "kros" )  
4.   (buy <kris> <cd>)  
5.   (setf (rate-for promo) <cd> 0.00)  
    (buy <kris> <cd>)))
```



```
(at  
  ((gf-call buy ?user ?article)  
   (most-recent (gf-call login ?user _ _ _)  
                 (rate-for ?rate ?article)))  
  (format t "~s gets a ~s % discount on ~s" ?user ?rate ?article))
```

# Stateful context-exposure

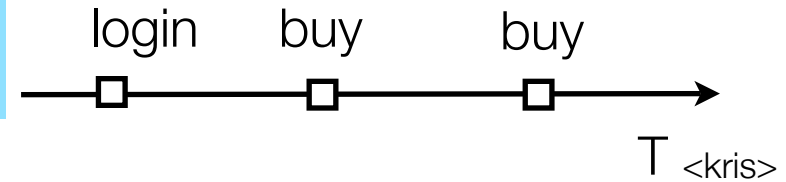
```
1. (let ((promo (singleton-instance 'promotions)))  
2.   (setf (rate-for promo) <cd> 0.05)  
3.   (login <kris> <shop> "kris" "kros" )  
4.   (buy <kris> <cd>)  
5.   (setf (rate-for promo) <cd> 0.00)  
6.   (buy <kris> <cd>)))
```



```
(at  
  ((gf-call buy ?user ?article)  
   (most-recent (gf-call login ?user _ _ _)  
                 (rate-for ?rate ?article)))  
  (format t "~s gets a ~s % discount on ~s" ?user ?rate ?article))
```

# Stateful context-exposure

```
1. (let ((promo (singleton-instance 'promotions)))  
2.   (setf (rate-for promo) <cd> 0.05)  
3.   (login <kris> <shop> "kris" "kros" )  
4.   (buy <kris> <cd>)  
5.   (setf (rate-for promo) <cd> 0.00)  
6.   (buy <kris> <cd>)))
```

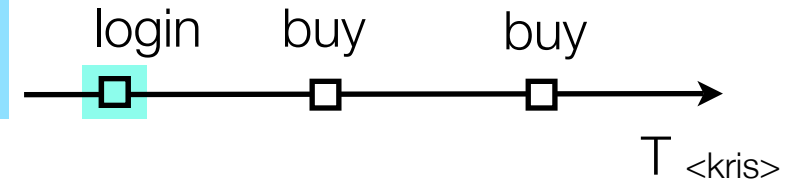


```
(at  
  ((gf-call buy ?user ?article)  
   (most-recent (gf-call login ?user _ _ _)  
                 (rate-for ?rate ?article)))  
  (format t "~s gets a ~s % discount on ~s" ?user ?rate ?article))
```



# Stateful context-exposure

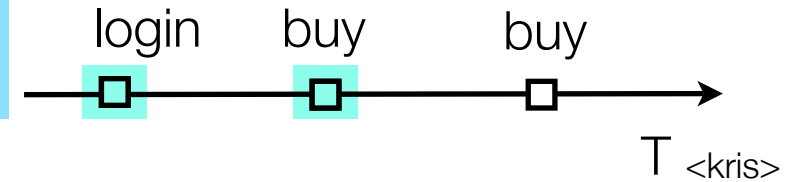
```
1. (let ((promo (singleton-instance 'promotions)))  
2.   (setf (rate-for promo) <cd> 0.05)  
3.   (login <kris> <shop> "kris" "kros" )  
4.   (buy <kris> <cd>)  
5.   (setf (rate-for promo) <cd> 0.00)  
6.   (buy <kris> <cd>)))
```



```
(at  
  ((gf-call buy ?user ?article)  
   (most-recent (gf-call login ?user _ _ _)  
                 (rate-for ?rate ?article)))  
  (format t "~s gets a ~s % discount on ~s" ?user ?rate ?article))
```

# Stateful context-exposure

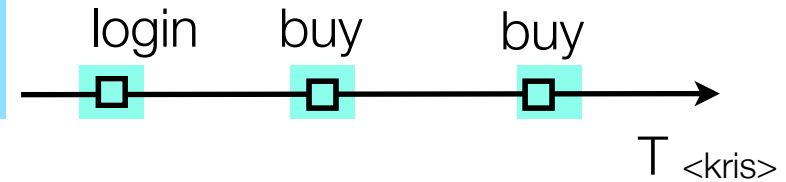
```
1. (let ((promo (singleton-instance 'promotions)))
2.   (setf (rate-for promo) <cd> 0.05)
3.   (login <kris> <shop> "kris" "kros" )
4.   (buy <kris> <cd>)
5.   (setf (rate-for promo) <cd> 0.00)
6.   (buy <kris> <cd>)))
```



```
(at
  ((gf-call buy ?user ?article)
   (most-recent (gf-call login ?user _ _ _)
                 (rate-for ?rate ?article)))
  (format t "~s gets a ~s % discount on ~s" ?user ?rate ?article))
```

# Stateful context-exposure

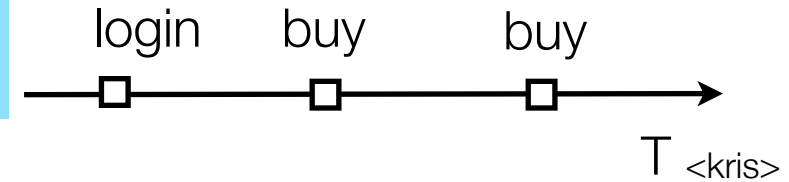
```
1. (let ((promo (singleton-instance 'promotions)))  
2.   (setf (rate-for promo) <cd> 0.05)  
3.   (login <kris> <shop> "kris" "kros" )  
4.   (buy <kris> <cd>)  
5.   (setf (rate-for promo) <cd> 0.00)  
6.   (buy <kris> <cd>)))
```



```
(at  
  ((gf-call buy ?user ?article)  
   (most-recent (gf-call login ?user _ _ _)  
                 (rate-for ?rate ?article)))  
  (format t "~s gets a ~s % discount on ~s" ?user ?rate ?article))
```

# Stateful context-exposure

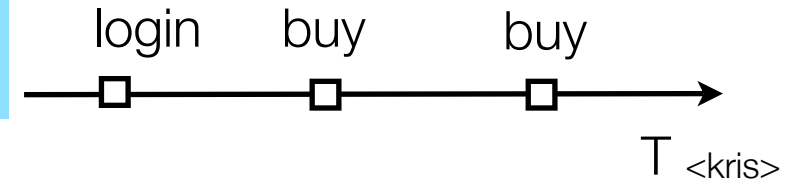
```
1. (let ((promo (singleton-instance 'promotions)))  
2.   (setf (rate-for promo) <cd> 0.05)  
3.   (login <kris> <shop> "kris" "kros" )  
4.   (buy <kris> <cd>)  
5.   (setf (rate-for promo) <cd> 0.00)  
6.   (buy <kris> <cd>)))
```



```
(at  
  ((gf-call buy ?user ?article)  
   (most-recent (gf-call login ?user _ _ _)  
                 (rate-for ?rate ?article)))  
  (format t "~s gets a ~s % discount on ~s" ?user ?rate ?article))
```

# Stateful context-exposure

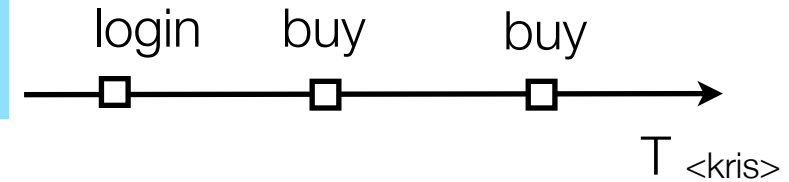
```
1. (let ((promo (singleton-instance 'promotions)))  
2.   (setf (rate-for promo) <cd> 0.05)  
3.   (login <kris> <shop> "kris" "kros" )  
4.   (buy <kris> <cd>)  
5.   (setf (rate-for promo) <cd> 0.00)  
6.   (buy <kris> <cd>)))
```



```
(at  
  ((gf-call buy ?user ?article)  
   (most-recent (gf-call login ?user _ _ _)  
                 (rate-for ?rate ?article)))  
  (format t "~s gets a ~s % discount on ~s" ?user ?rate ?article))
```

# Stateful context-exposure

```
1. (let ((promo (singleton-instance 'promotions)))
2.   (setf (rate-for promo) <cd> 0.05)
3.   (login <kris> <shop> "kris" "kros" )
4.   (buy <kris> <cd>)
5.   (setf (rate-for promo) <cd> 0.00)
6.   (buy <kris> <cd>)))
```

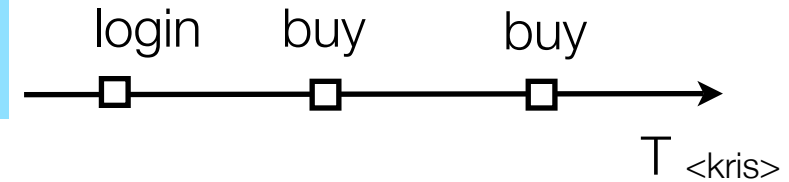


```
(at
  ((gf-call buy ?user ?article)
   (most-recent (gf-call login ?user _ _ _ )
                 (rate-for ?rate ?article)))
  (format t "~s gets a ~s % discount on ~s" ?user ?rate ?article))
```

```
(defrule (rate-for ?rate ?article)
  (escape ?promo (singleton-instance 'promotions)
  (escape ?rate (discount-rate-for ?promo ?article)))
```

# Stateful context-exposure

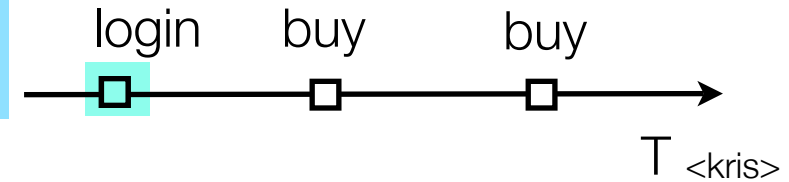
```
1. (let ((promo (singleton-instance 'promotions)))  
2.   (setf (rate-for promo) <cd> 0.05)  
3.   (login <kris> <shop> "kris" "kros" )  
4.   (buy <kris> <cd>)  
5.   (setf (rate-for promo) <cd> 0.00)  
6.   (buy <kris> <cd>)))
```



```
(at  
  ((gf-call buy ?user ?article)  
   (most-recent (gf-call login ?user _ _ _)  
                 (rate-for ?rate ?article)))  
  (format t "~s gets a ~s % discount on ~s" ?user ?rate ?article))
```

# Stateful context-exposure

```
1. (let ((promo (singleton-instance 'promotions)))
2.   (setf (rate-for promo) <cd> 0.05)
3.   (login <kris> <shop> "kris" "kros" )
4.   (buy <kris> <cd>)
5.   (setf (rate-for promo) <cd> 0.00)
6.   (buy <kris> <cd>)))
```

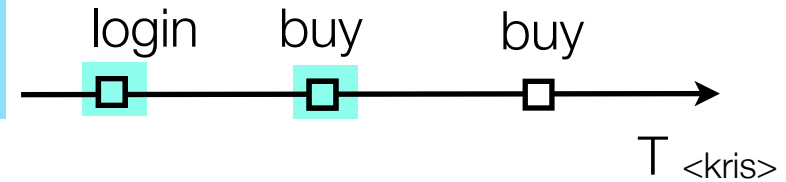


```
(at
  ((gf-call buy ?user ?article)
   (most-recent (gf-call login ?user _ _ _)
                 (rate-for ?rate ?article)))
  (format t "~s gets a ~s % discount on ~s" ?user ?rate ?article))
```



# Stateful context-exposure

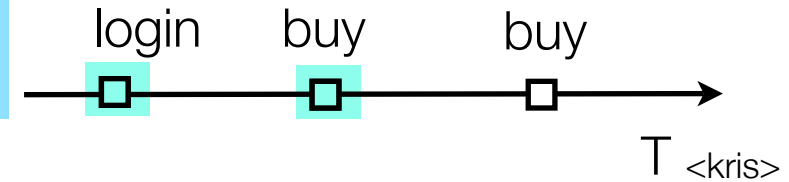
```
1. (let ((promo (singleton-instance 'promotions)))
2.   (setf (rate-for promo) <cd> 0.05)
3.   (login <kris> <shop> "kris" "kros" )
4.   (buy <kris> <cd>)
5.   (setf (rate-for promo) <cd> 0.00)
6.   (buy <kris> <cd>)))
```



```
(at
  ((gf-call buy ?user ?article)
   (most-recent (gf-call login ?user _ _ _)
                 (rate-for ?rate ?article)))
  (format t "~s gets a ~s % discount on ~s" ?user ?rate ?article))
```

# Stateful context-exposure

```
1. (let ((promo (singleton-instance 'promotions)))  
2.   (setf (rate-for promo) <cd> 0.05)  
3.   (login <kris> <shop> "kris" "kros" )  
4.   (buy <kris> <cd>)  
5.   (setf (rate-for promo) <cd> 0.00)  
6.   (buy <kris> <cd>)))
```

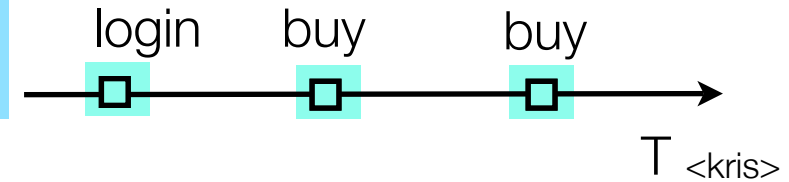


```
(at  
  ((gf-call buy ?user ?article)  
   (most-recent (gf-call login ?user _ _ _)  
                 (rate-for ?rate ?article)))  
  (format t "~s gets a ~s % discount on ~s" ?user ?rate ?article))
```

```
"<kris> gets a 0.05% discount on <cd>"
```

# Stateful context-exposure

```
1. (let ((promo (singleton-instance 'promotions)))
2.   (setf (rate-for promo) <cd> 0.05)
3.   (login <kris> <shop> "kris" "kros" )
4.   (buy <kris> <cd>)
5.   (setf (rate-for promo) <cd> 0.00)
6.   (buy <kris> <cd>)))
```

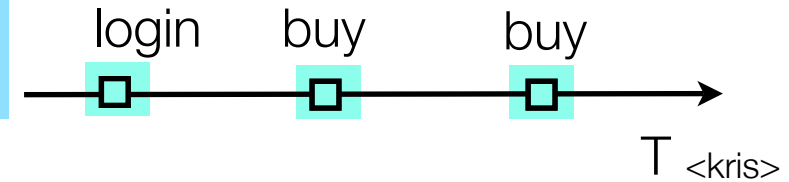


```
(at
  ((gf-call buy ?user ?article)
   (most-recent (gf-call login ?user _ _ _ )
                 (rate-for ?rate ?article)))
  (format t "~s gets a ~s % discount on ~s" ?user ?rate ?article))
```

```
"<kris> gets a 0.05% discount on <cd>"
```

# Stateful context-exposure

```
1. (let ((promo (singleton-instance 'promotions)))
2.   (setf (rate-for promo) <cd> 0.05)
3.   (login <kris> <shop> "kris" "kros" )
4.   (buy <kris> <cd>)
5.   (setf (rate-for promo) <cd> 0.00)
6.   (buy <kris> <cd>)))
```



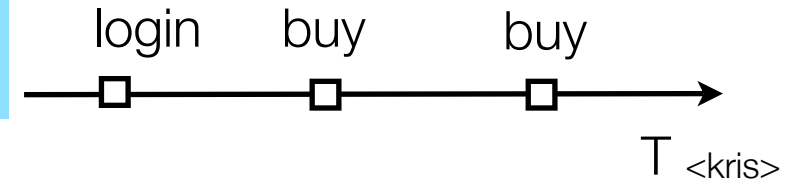
```
(at
  ((gf-call buy ?user ?article)
   (most-recent (gf-call login ?user _ _ _)
                 (rate-for ?rate ?article)))
  (format t "~s gets a ~s % discount on ~s" ?user ?rate ?article))
```

"<kris> gets a 0.05% discount on <cd>"

"<kris> gets a 0.05% discount on <cd>"

# Stateful context-exposure

```
1. (let ((promo (singleton-instance 'promotions)))  
2.   (setf (rate-for promo) <cd> 0.05)  
3.   (login <kris> <shop> "kris" "kros" )  
4.   (buy <kris> <cd>)  
5.   (setf (rate-for promo) <cd> 0.00)  
6.   (buy <kris> <cd>)))
```



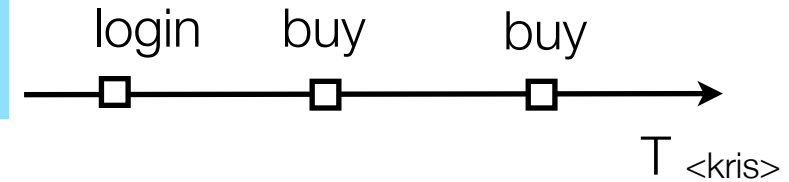
```
(at  
  ((gf-call buy ?user ?article)  
   (most-recent (gf-call login ?user _ _ _)  
                 (rate-for ?rate ?article)))  
  (format t "~s gets a ~s % discount on ~s" ?user ?rate ?article))
```

"<kris> gets a 0.05% discount on <cd>"

"<kris> gets a 0.05% discount on <cd>"

# Stateful context-exposure

```
1. (let ((promo (singleton-instance 'promotions)))
2.   (setf (rate-for promo) <cd> 0.05)
3.   (login <kris> <shop> "kris" "kros" )
4.   (buy <kris> <cd>)
5.   (setf (rate-for promo) <cd> 0.00)
6.   (buy <kris> <cd>)))
```



```
(at
  ((gf-call buy ?user ?article)
   (most-recent (gf-call login ?user _ _ _)
                 (rate-for ?rate ?article)))
  (format t "~s gets a ~s % discount on ~s" ?user ?rate ?article))
```

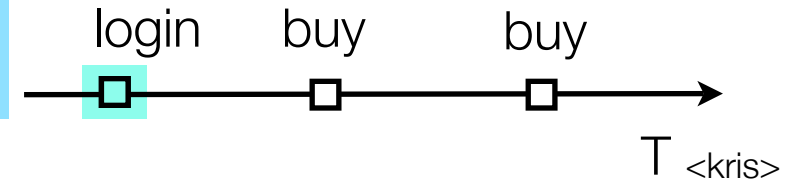
"<kris> gets a 0.05% discount on <cd>"

"<kris> gets a 0.05% discount on <cd>"

```
(at
  ((gf-call buy ?user ?article)
   (rate-for ?rate ?article)
   (most-recent (gf-call login ?user _ _ _)))
  (format t "~s gets a ~s % discount on ~s" ?user ?rate ?article))
```

# Stateful context-exposure

```
1. (let ((promo (singleton-instance 'promotions)))
2.   (setf (rate-for promo) <cd> 0.05)
3.   (login <kris> <shop> "kris" "kros" )
4.   (buy <kris> <cd>)
5.   (setf (rate-for promo) <cd> 0.00)
6.   (buy <kris> <cd>)))
```



```
(at
  ((gf-call buy ?user ?article)
   (most-recent (gf-call login ?user _ _ _)
                 (rate-for ?rate ?article)))
  (format t "~s gets a ~s % discount on ~s" ?user ?rate ?article))
```

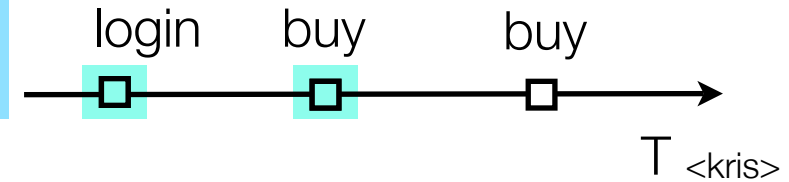
"<kris> gets a 0.05% discount on <cd>"

"<kris> gets a 0.05% discount on <cd>"

```
(at
  ((gf-call buy ?user ?article)
   (rate-for ?rate ?article)
   (most-recent (gf-call login ?user _ _ _)))
  (format t "~s gets a ~s % discount on ~s" ?user ?rate ?article))
```

# Stateful context-exposure

```
1. (let ((promo (singleton-instance 'promotions)))
2.   (setf (rate-for promo) <cd> 0.05)
3.   (login <kris> <shop> "kris" "kros" )
4.   (buy <kris> <cd>)
5.   (setf (rate-for promo) <cd> 0.00)
6.   (buy <kris> <cd>)))
```



```
(at
  ((gf-call buy ?user ?article)
   (most-recent (gf-call login ?user _ _ _)
                 (rate-for ?rate ?article)))
  (format t "~s gets a ~s % discount on ~s" ?user ?rate ?article))
```

"<kris> gets a 0.05% discount on <cd>"

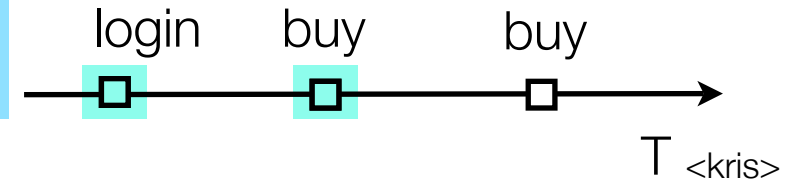
"<kris> gets a 0.05% discount on <cd>"

```
(at
  ((gf-call buy ?user ?article)
   (rate-for ?rate ?article)
   (most-recent (gf-call login ?user _ _ _)))
  (format t "~s gets a ~s % discount on ~s" ?user ?rate ?article))
```



# Stateful context-exposure

```
1. (let ((promo (singleton-instance 'promotions)))
2.   (setf (rate-for promo) <cd> 0.05)
3.   (login <kris> <shop> "kris" "kros" )
4.   (buy <kris> <cd>)
5.   (setf (rate-for promo) <cd> 0.00)
6.   (buy <kris> <cd>)))
```



```
(at
  ((gf-call buy ?user ?article)
   (most-recent (gf-call login ?user _ _ _)
                 (rate-for ?rate ?article)))
  (format t "~s gets a ~s % discount on ~s" ?user ?rate ?article))
```

"<kris> gets a 0.05% discount on <cd>"

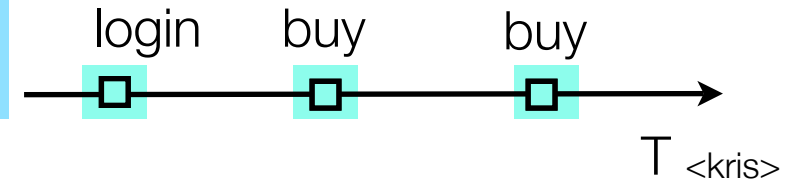
"<kris> gets a 0.05% discount on <cd>"

```
(at
  ((gf-call buy ?user ?article)
   (rate-for ?rate ?article)
   (most-recent (gf-call login ?user _ _ _)))
  (format t "~s gets a ~s % discount on ~s" ?user ?rate ?article))
```

"<kris> gets a 0.05% discount on <cd>"

# Stateful context-exposure

```
1. (let ((promo (singleton-instance 'promotions)))
2.   (setf (rate-for promo) <cd> 0.05)
3.   (login <kris> <shop> "kris" "kros" )
4.   (buy <kris> <cd>)
5.   (setf (rate-for promo) <cd> 0.00)
6.   (buy <kris> <cd>)))
```



```
(at
  ((gf-call buy ?user ?article)
   (most-recent (gf-call login ?user _ _ _)
                 (rate-for ?rate ?article)))
  (format t "~s gets a ~s % discount on ~s" ?user ?rate ?article))
```

"<kris> gets a 0.05% discount on <cd>"

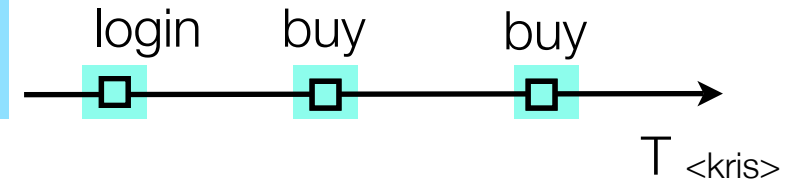
"<kris> gets a 0.05% discount on <cd>"

```
(at
  ((gf-call buy ?user ?article)
   (rate-for ?rate ?article)
   (most-recent (gf-call login ?user _ _ _)))
  (format t "~s gets a ~s % discount on ~s" ?user ?rate ?article))
```

"<kris> gets a 0.05% discount on <cd>"

# Stateful context-exposure

```
1. (let ((promo (singleton-instance 'promotions)))
2.   (setf (rate-for promo) <cd> 0.05)
3.   (login <kris> <shop> "kris" "kros" )
4.   (buy <kris> <cd>)
5.   (setf (rate-for promo) <cd> 0.00)
6.   (buy <kris> <cd>)))
```



```
(at
  ((gf-call buy ?user ?article)
   (most-recent (gf-call login ?user _ _ _ )
                 (rate-for ?rate ?article)))
  (format t "~s gets a ~s % discount on ~s" ?user ?rate ?article))
```

"<kris> gets a 0.05% discount on <cd>"

"<kris> gets a 0.05% discount on <cd>"

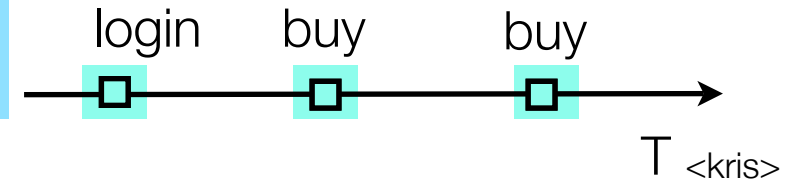
```
(at
  ((gf-call buy ?user ?article)
   (rate-for ?rate ?article)
   (most-recent (gf-call login ?user _ _ _ )))
  (format t "~s gets a ~s % discount on ~s" ?user ?rate ?article))
```

"<kris> gets a 0.05% discount on <cd>"

"<kris> gets a 0.00% discount on <cd>"

# Stateful context-exposure

```
1. (let ((promo (singleton-instance 'promotions)))
2.   (setf (rate-for promo) <cd> 0.05)
3.   (login <kris> <shop> "kris" "kros" )
4.   (buy <kris> <cd>)
5.   (setf (rate-for promo) <cd> 0.00)
6.   (buy <kris> <cd>)))
```



```
(at
  ((gf-call buy ?user ?article)
   (most-recent (gf-call login ?user _ _ _)
                 (rate-for ?rate ?article)))
  (format t "~s gets a ~s % discount on ~s" ?user ?rate ?article))
```

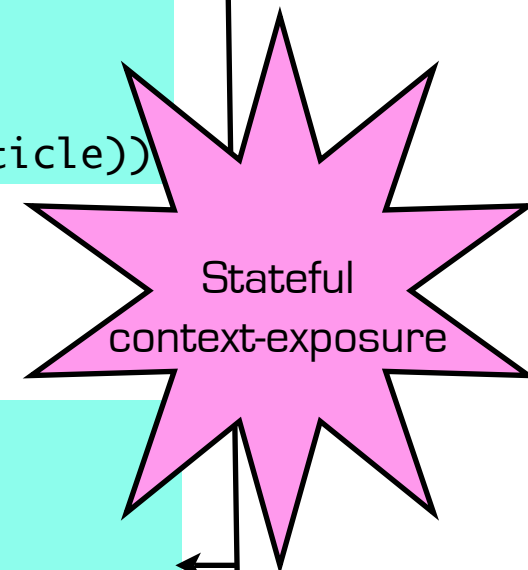
"<kris> gets a 0.05% discount on <cd>"

"<kris> gets a 0.05% discount on <cd>"

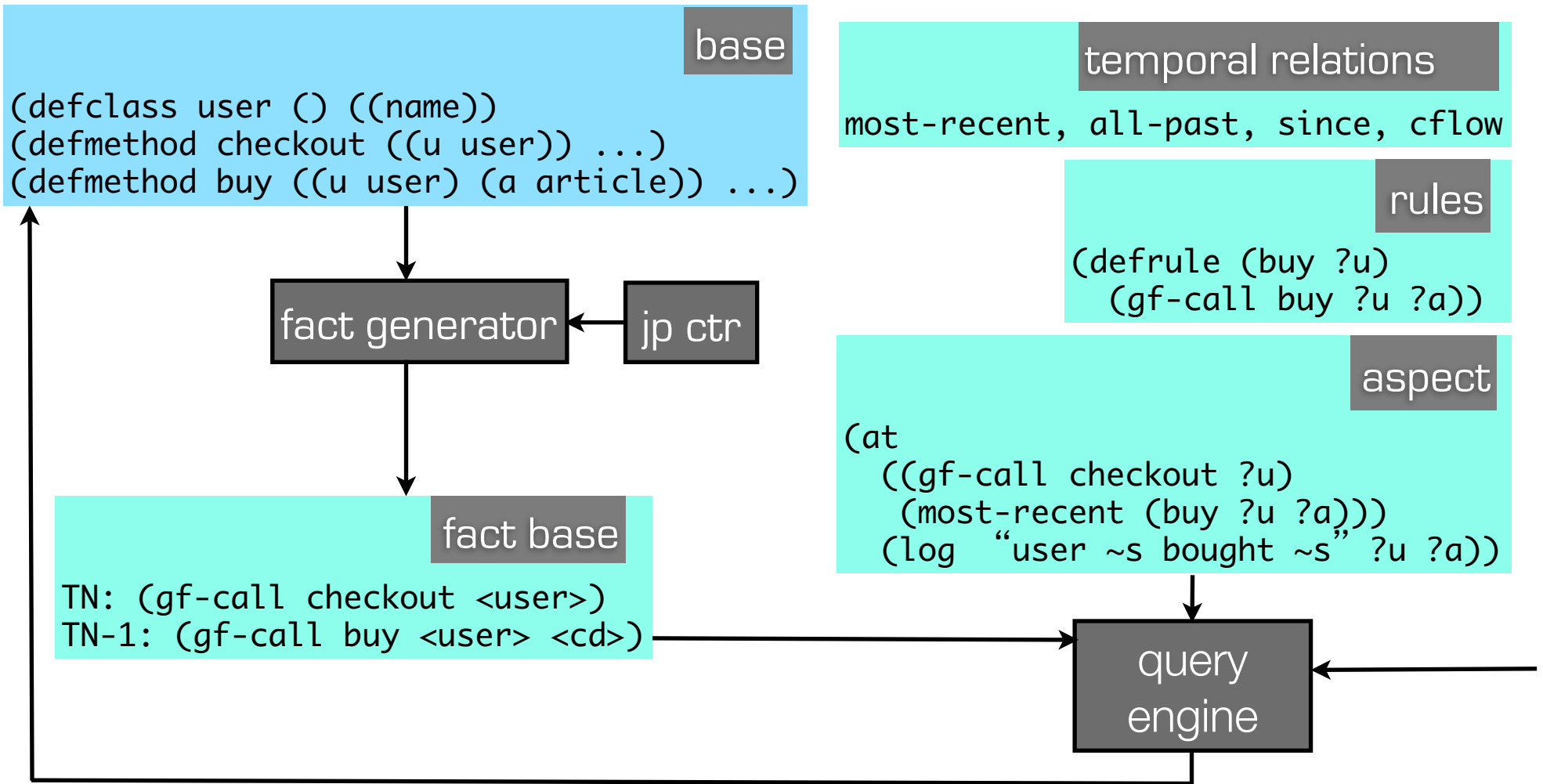
```
(at
  ((gf-call buy ?user ?article)
   (rate-for ?rate ?article)
   (most-recent (gf-call login ?user _ _ _)))
  (format t "~s gets a ~s % discount on ~s" ?user ?rate ?article))
```

"<kris> gets a 0.05% discount on <cd>"

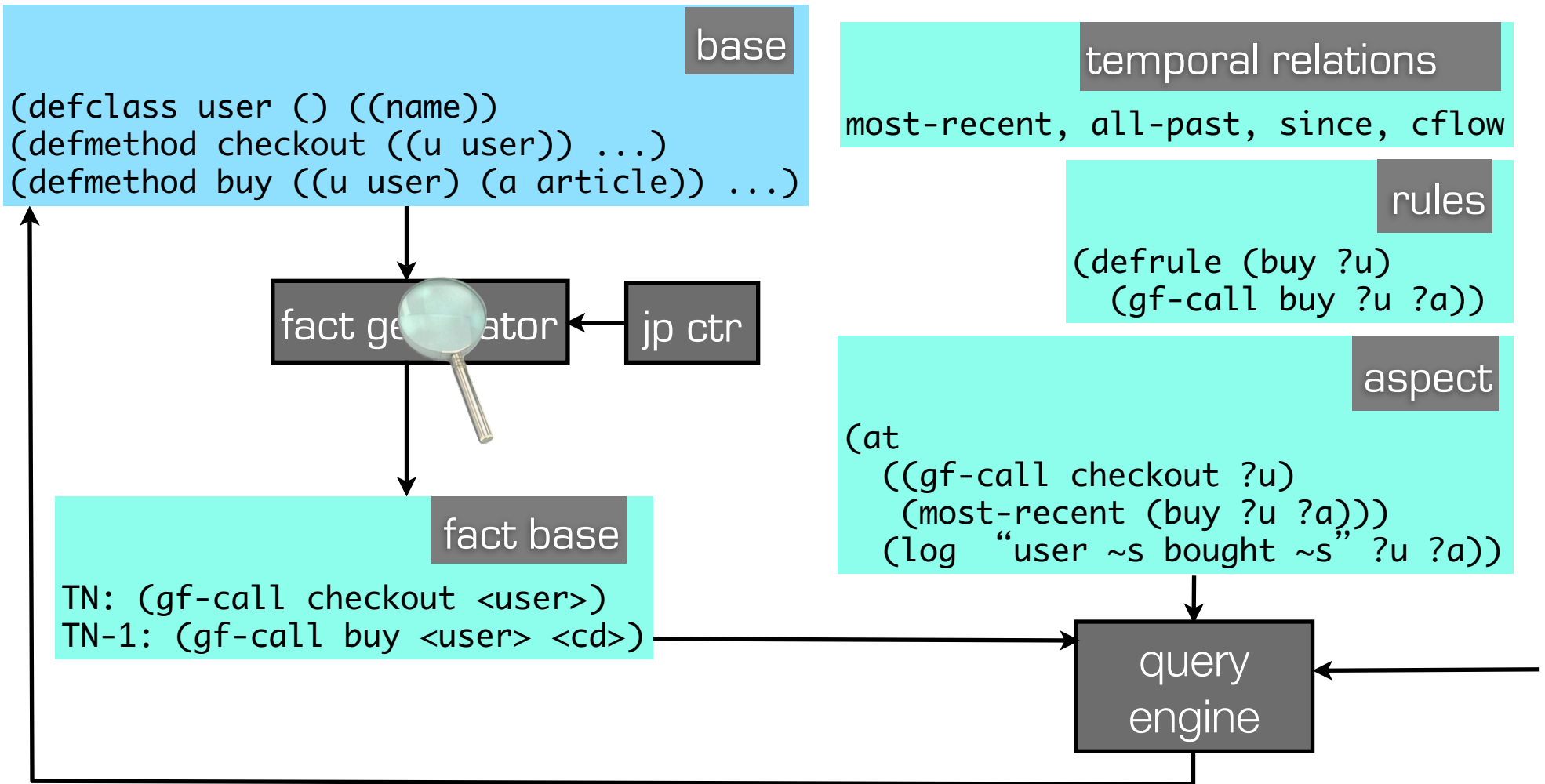
"<kris> gets a 0.00% discount on <cd>"



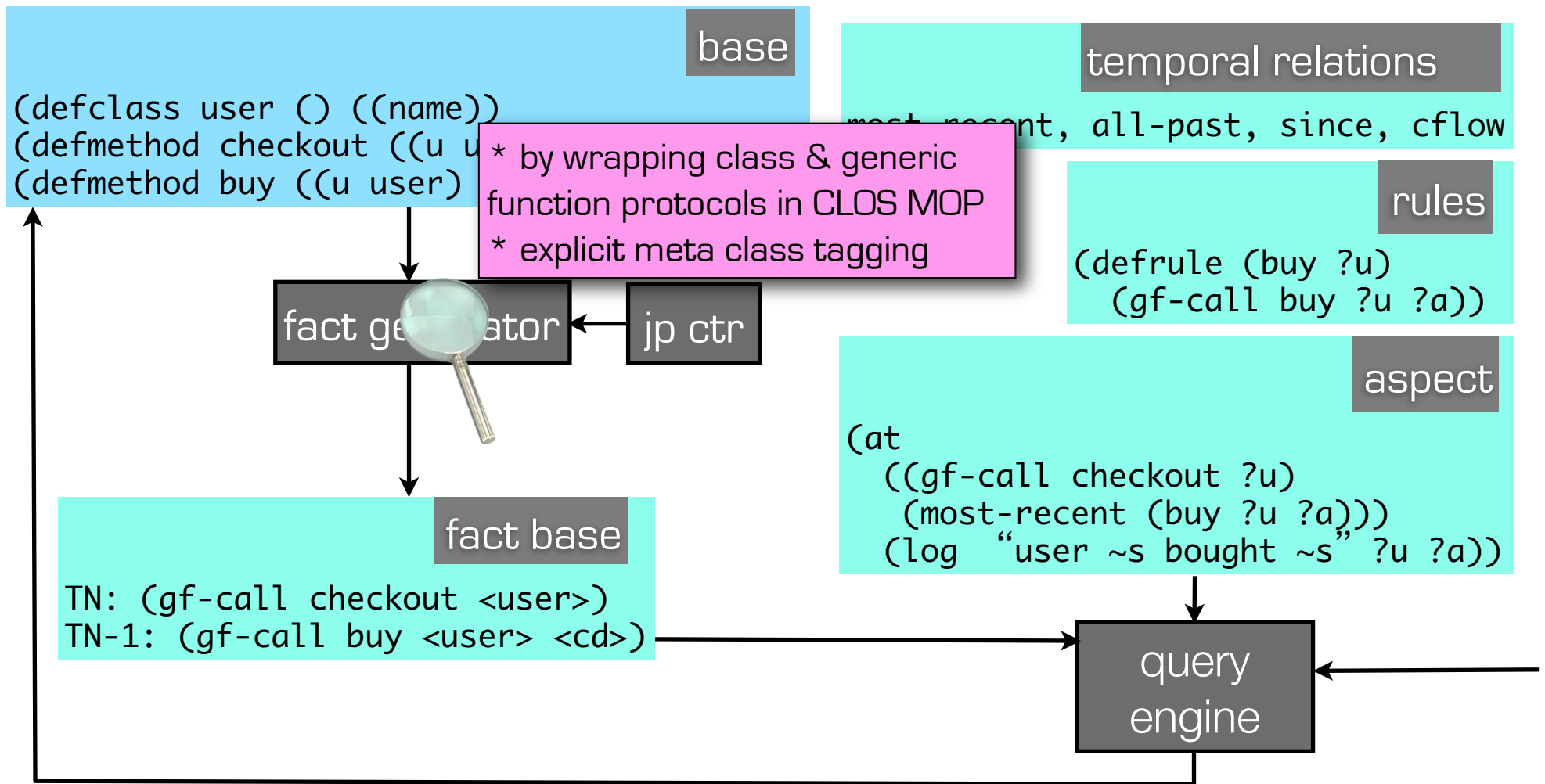
# Weaver Implementation



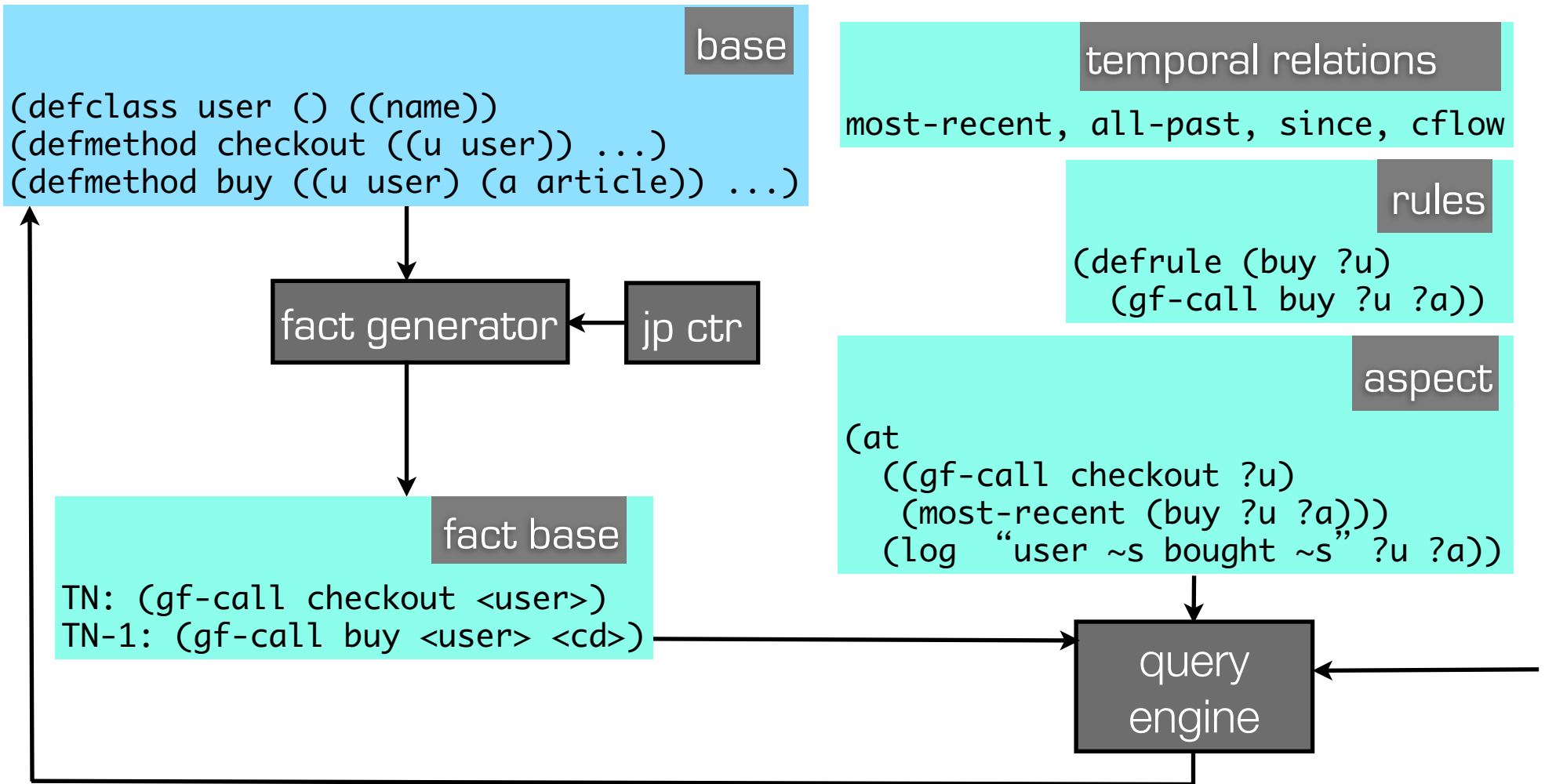
# Weaver Implementation



# Weaver Implementation

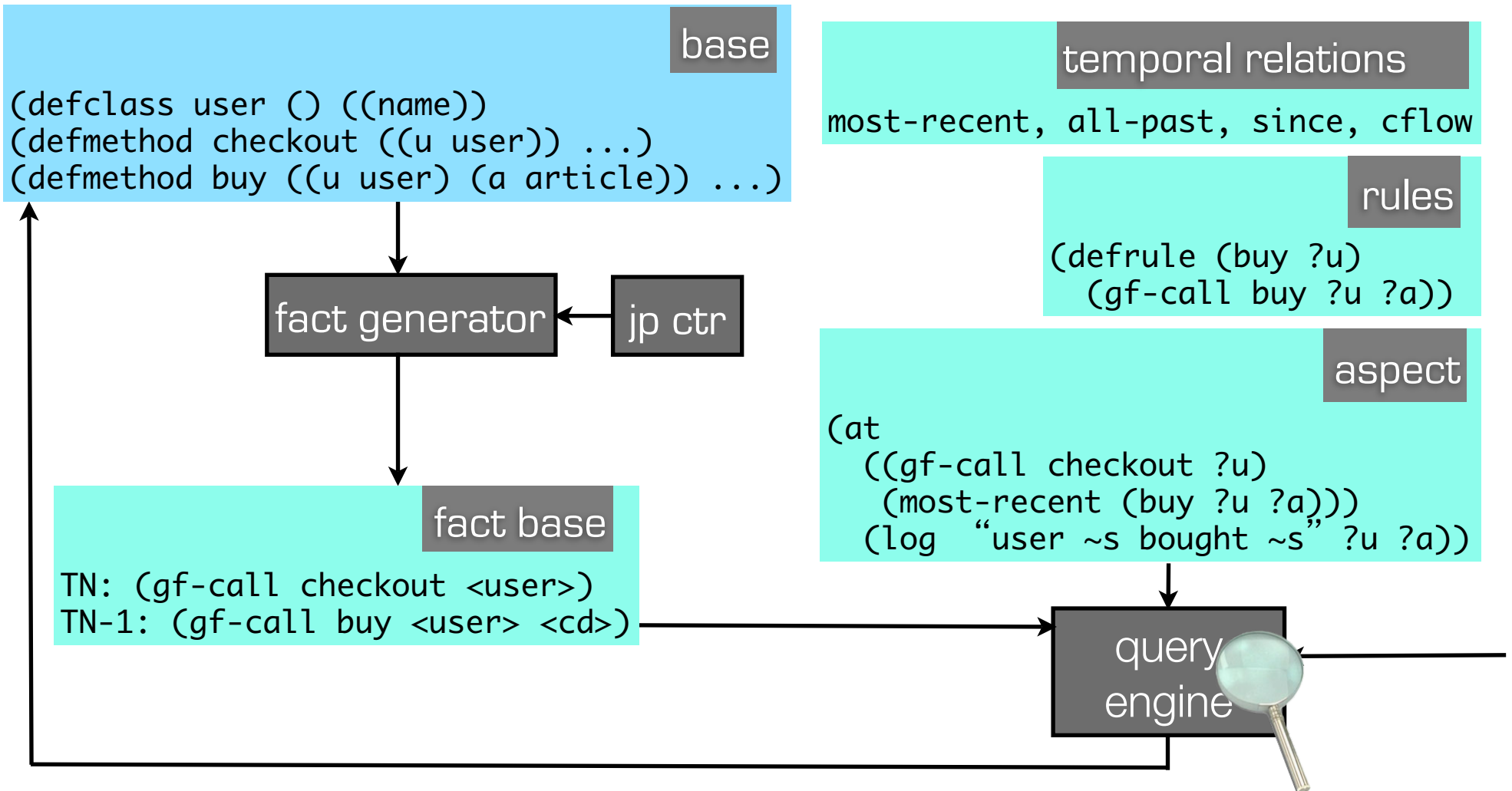


# Weaver Implementation

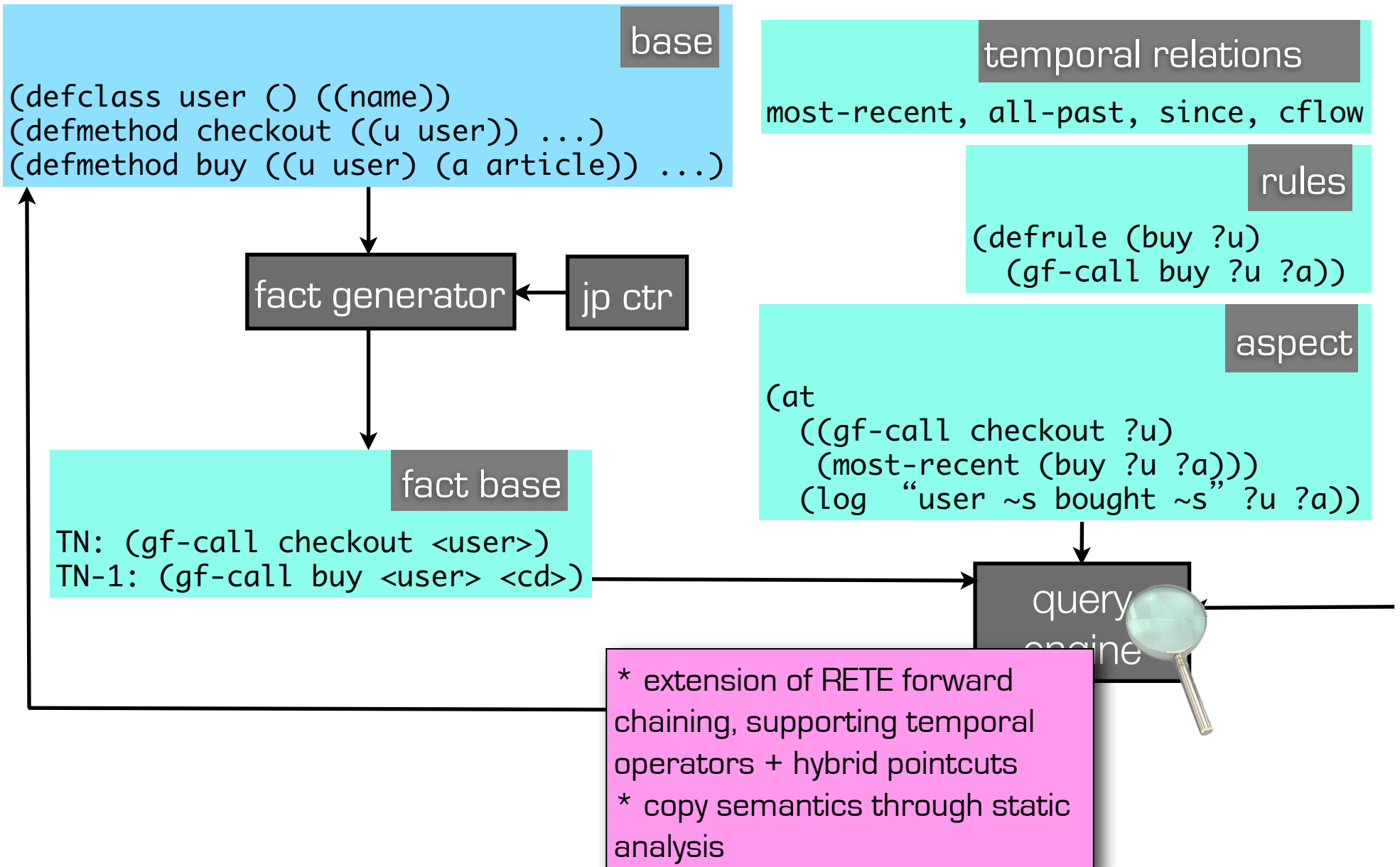




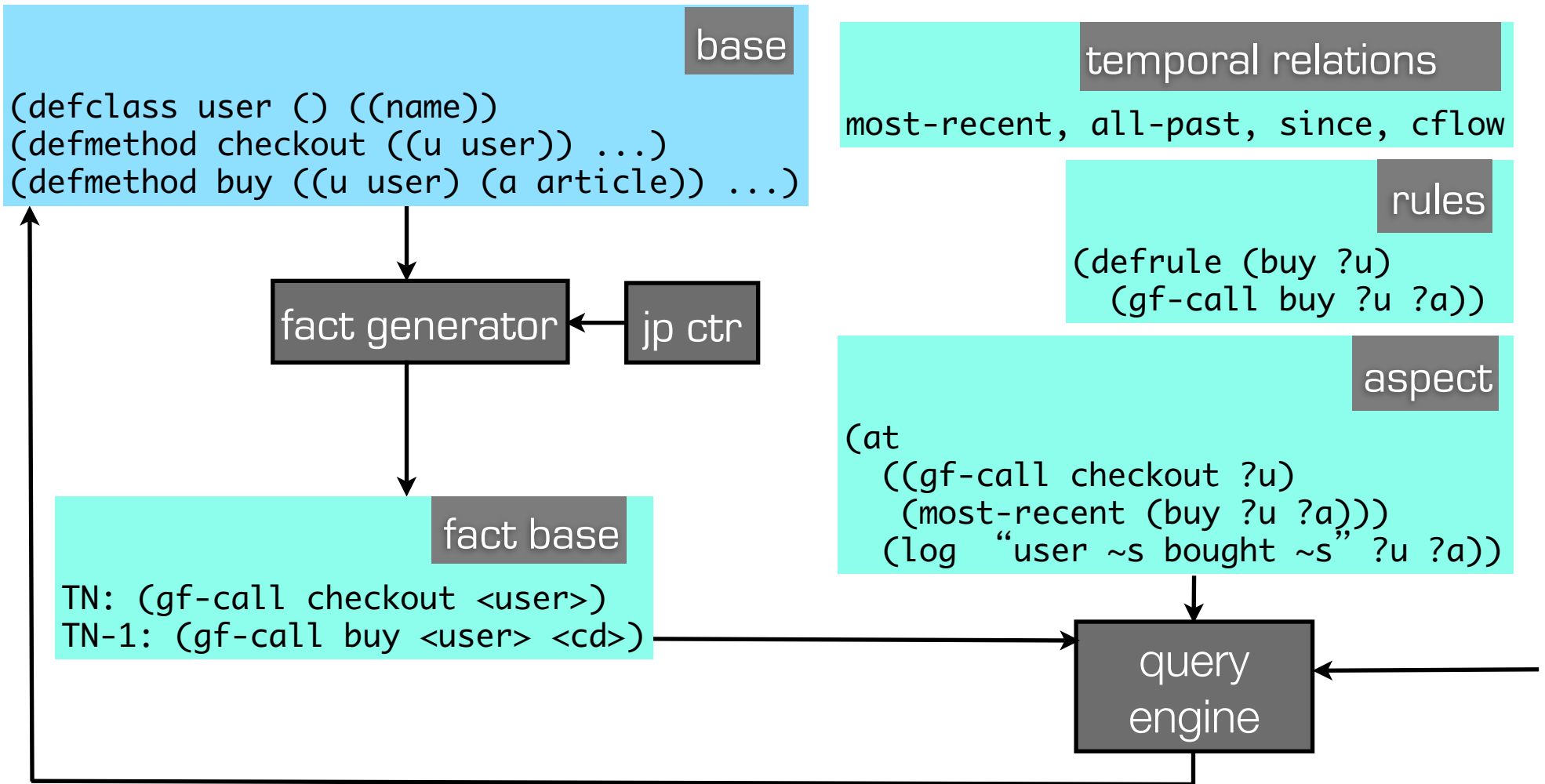
# Weaver Implementation



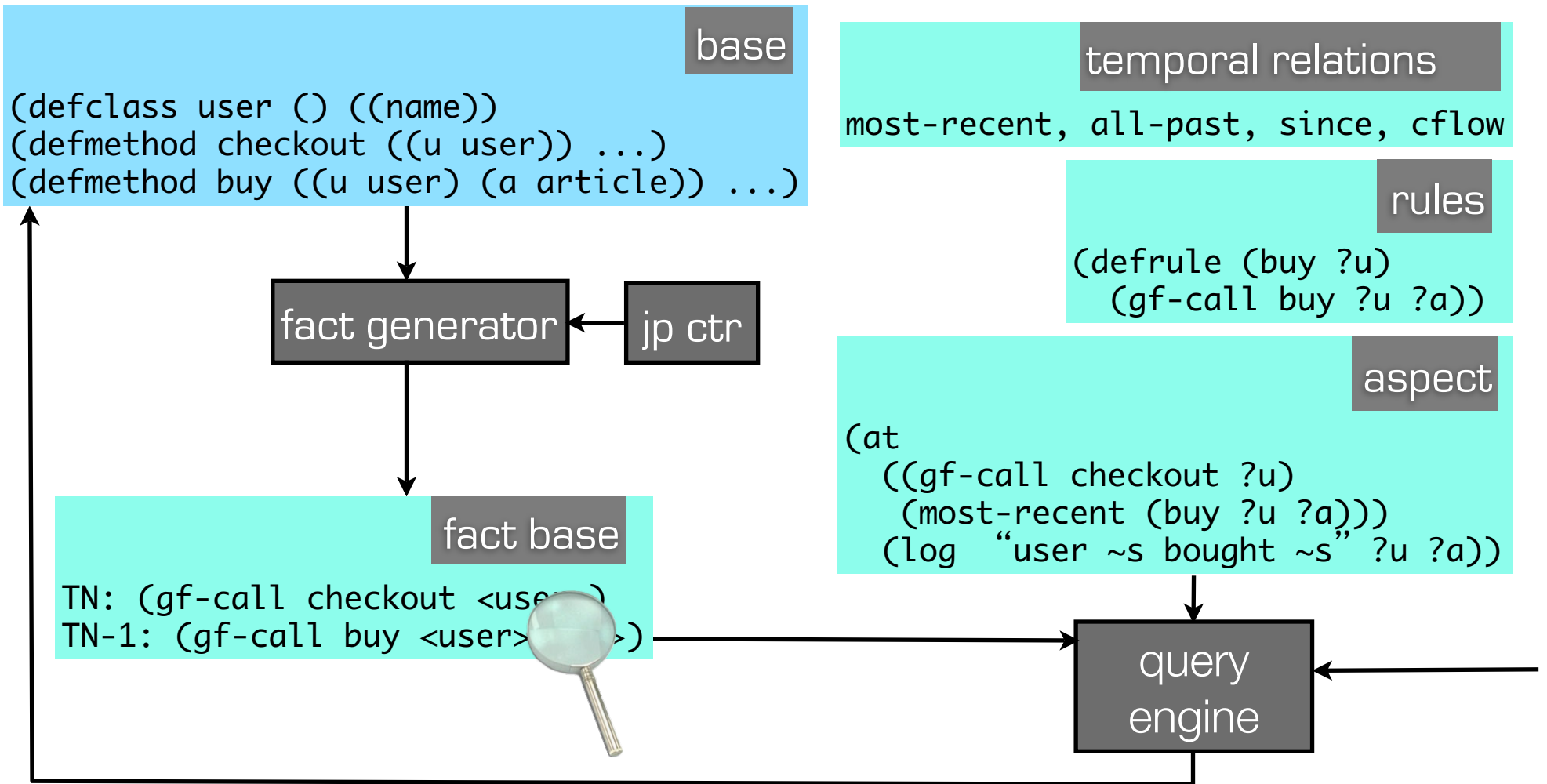
# Weaver Implementation



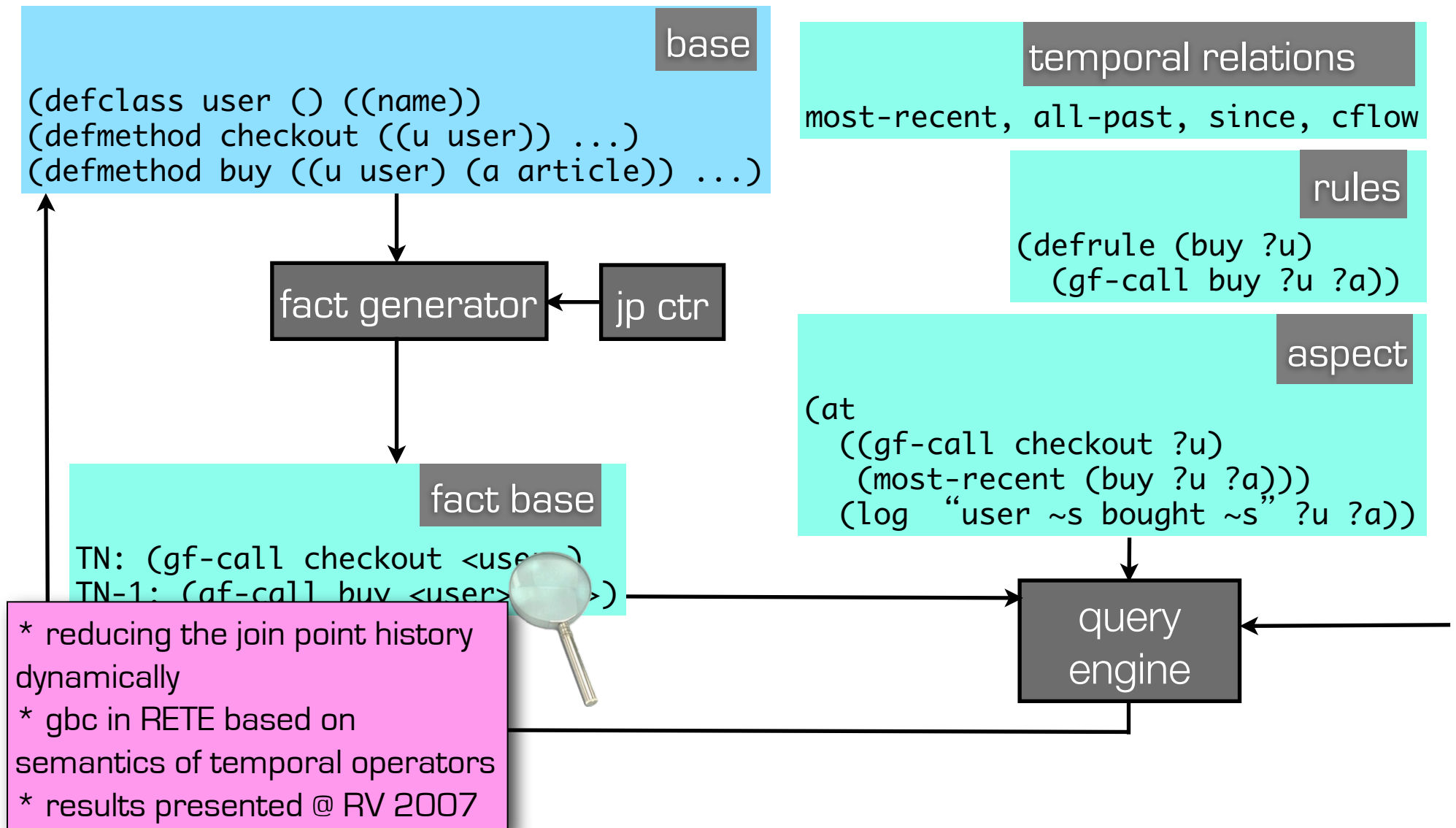
# Weaver Implementation



# Weaver Implementation



# Weaver Implementation

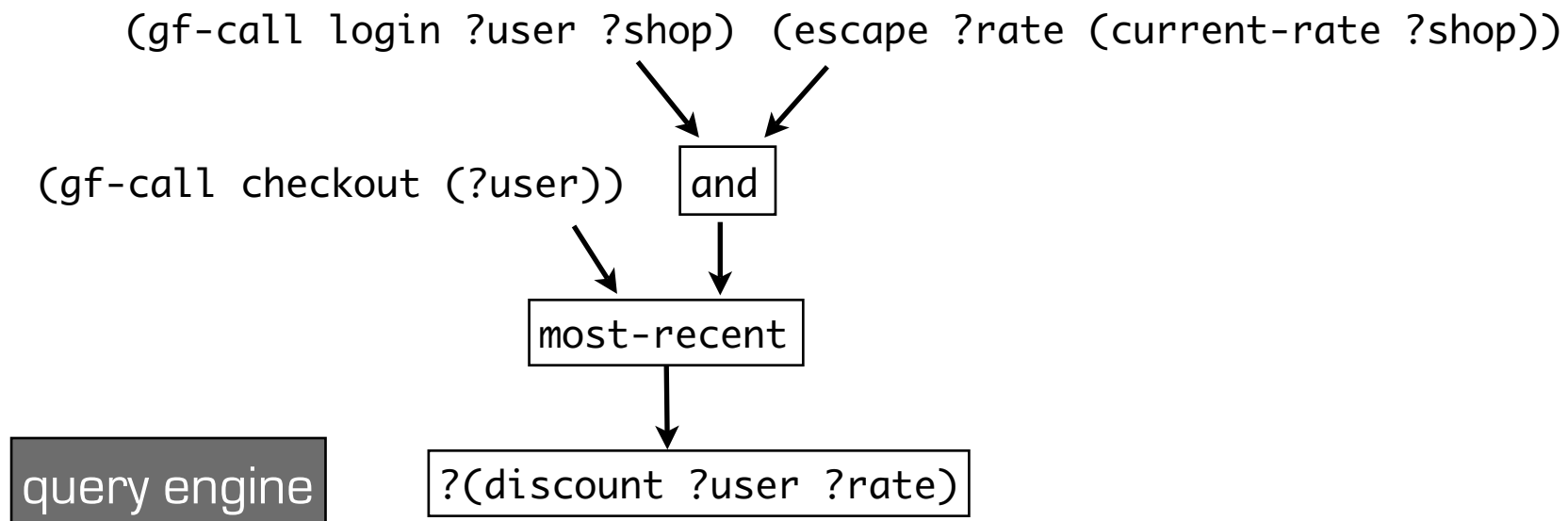


# Backward Chaining vs Rete Forward Chaining

```
(at ((gf-call checkout (?user))  
    (most-recent (gf-call login ?user ?shop)  
                  (escape ?rate (current-rate ?shop))))  
(discount ?user ?rate))
```

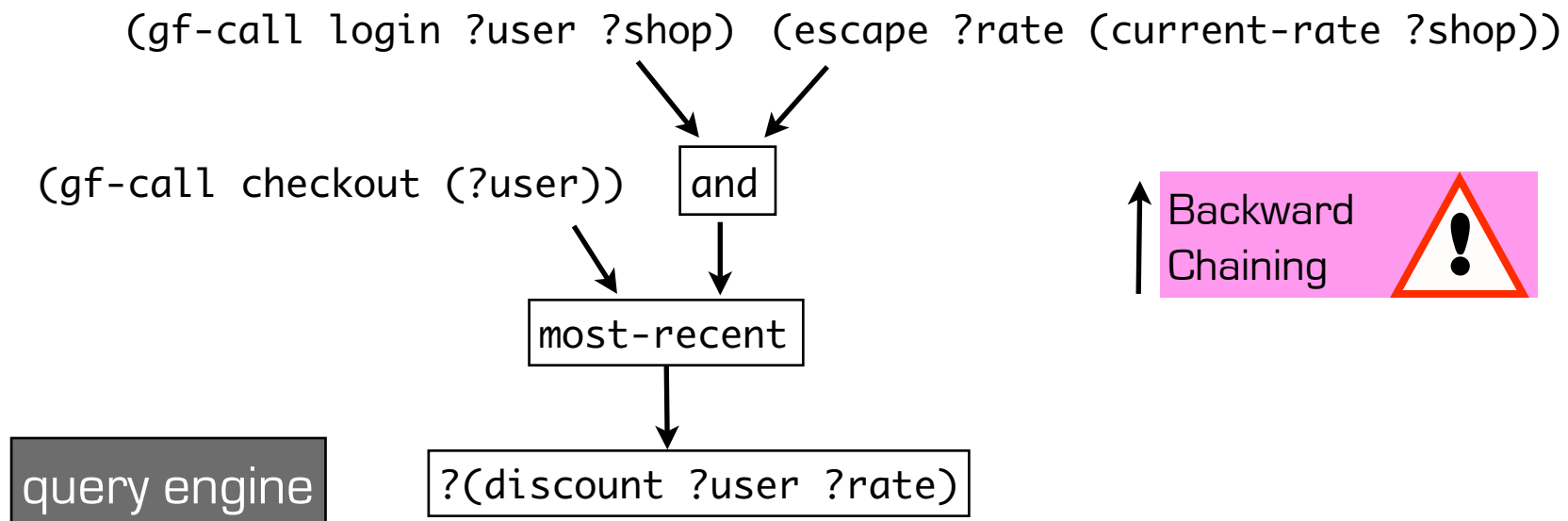
# Backward Chaining vs Rete Forward Chaining

```
(at ((gf-call checkout (?user))  
    (most-recent (gf-call login ?user ?shop)  
                  (escape ?rate (current-rate ?shop))))  
(discount ?user ?rate))
```



# Backward Chaining vs Rete Forward Chaining

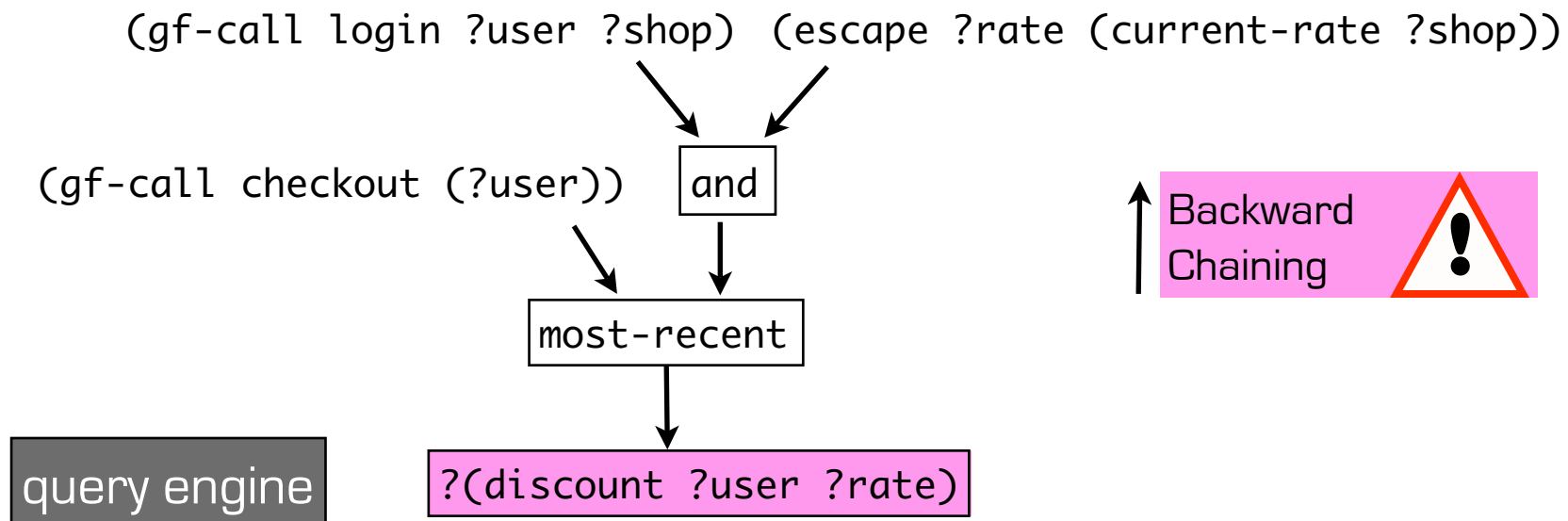
```
(at ((gf-call checkout (?user))  
    (most-recent (gf-call login ?user ?shop)  
                  (escape ?rate (current-rate ?shop)))))  
(discount ?user ?rate))
```





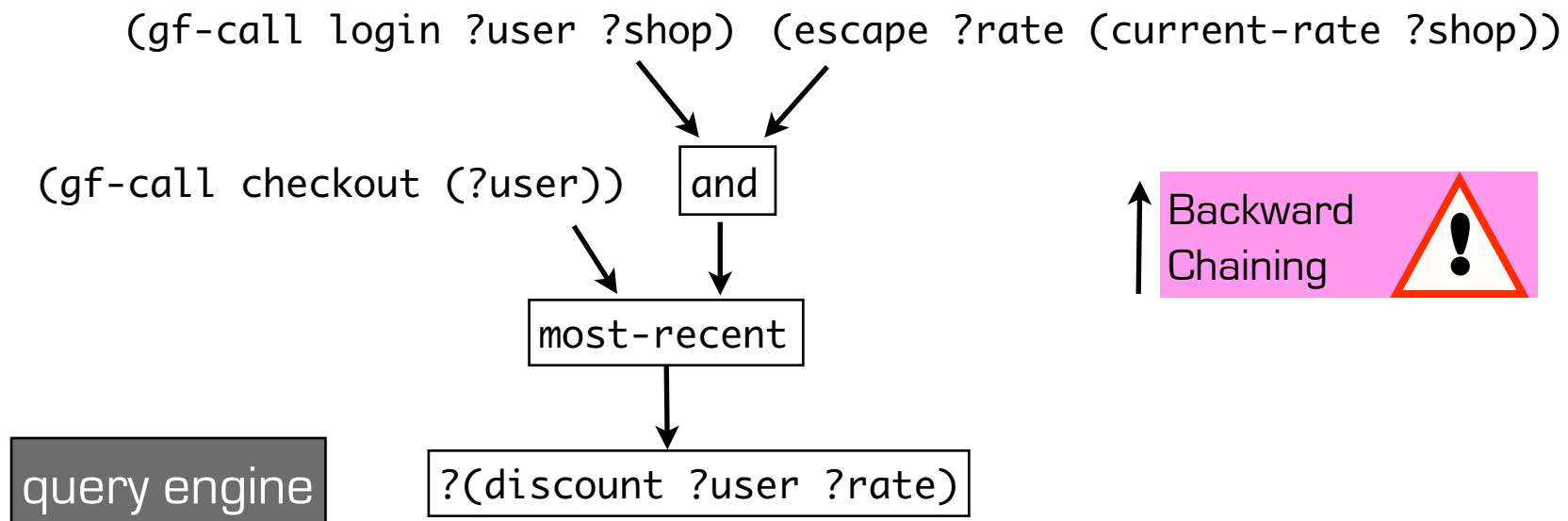
# Backward Chaining vs Rete Forward Chaining

```
(at ((gf-call checkout (?user))  
    (most-recent (gf-call login ?user ?shop)  
                  (escape ?rate (current-rate ?shop)))))  
(discount ?user ?rate))
```



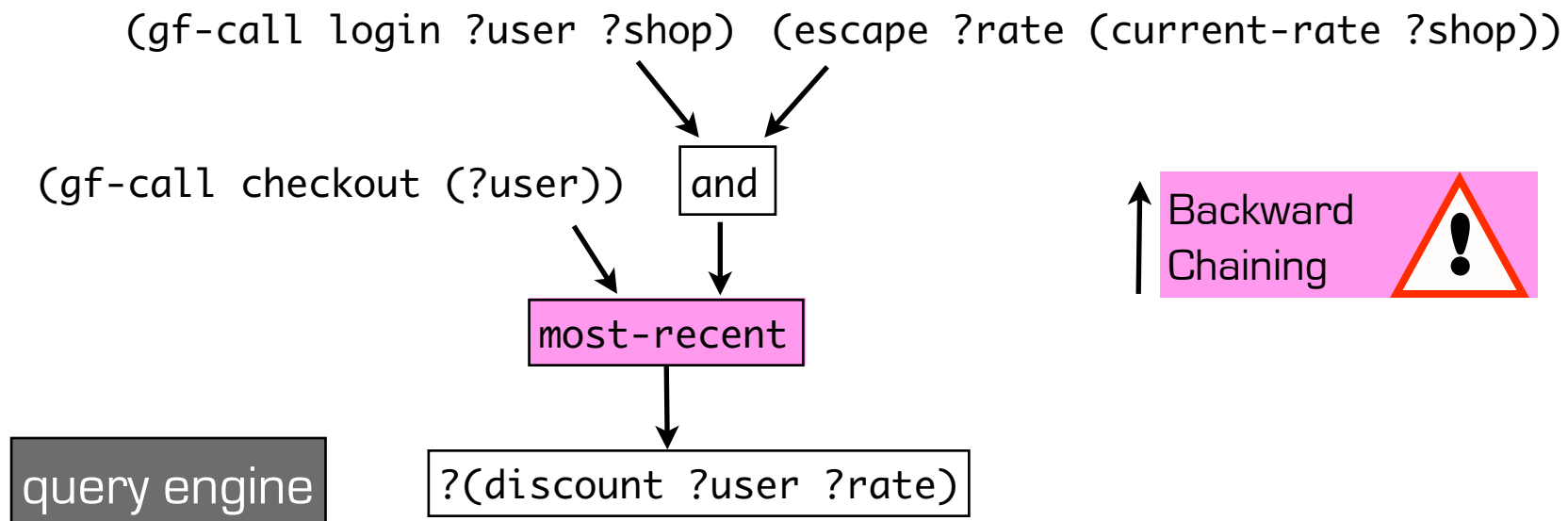
# Backward Chaining vs Rete Forward Chaining

```
(at ((gf-call checkout (?user))  
    (most-recent (gf-call login ?user ?shop)  
                  (escape ?rate (current-rate ?shop)))))  
(discount ?user ?rate))
```



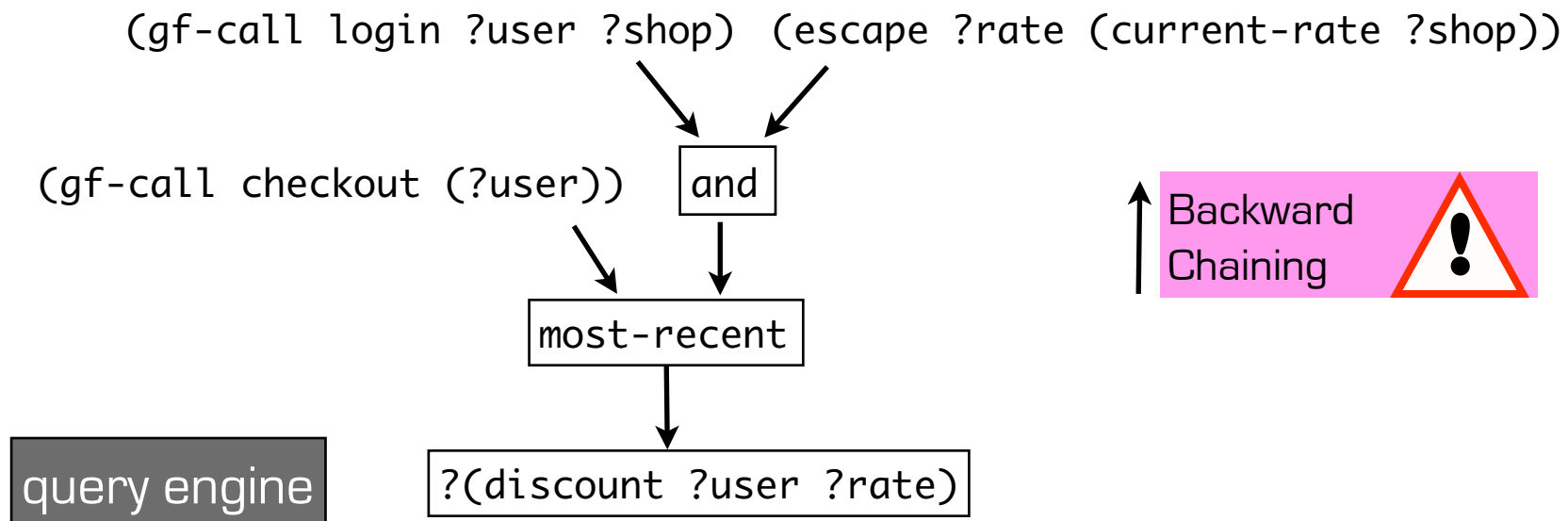
# Backward Chaining vs Rete Forward Chaining

```
(at ((gf-call checkout (?user))  
    (most-recent (gf-call login ?user ?shop)  
                  (escape ?rate (current-rate ?shop)))))  
(discount ?user ?rate))
```



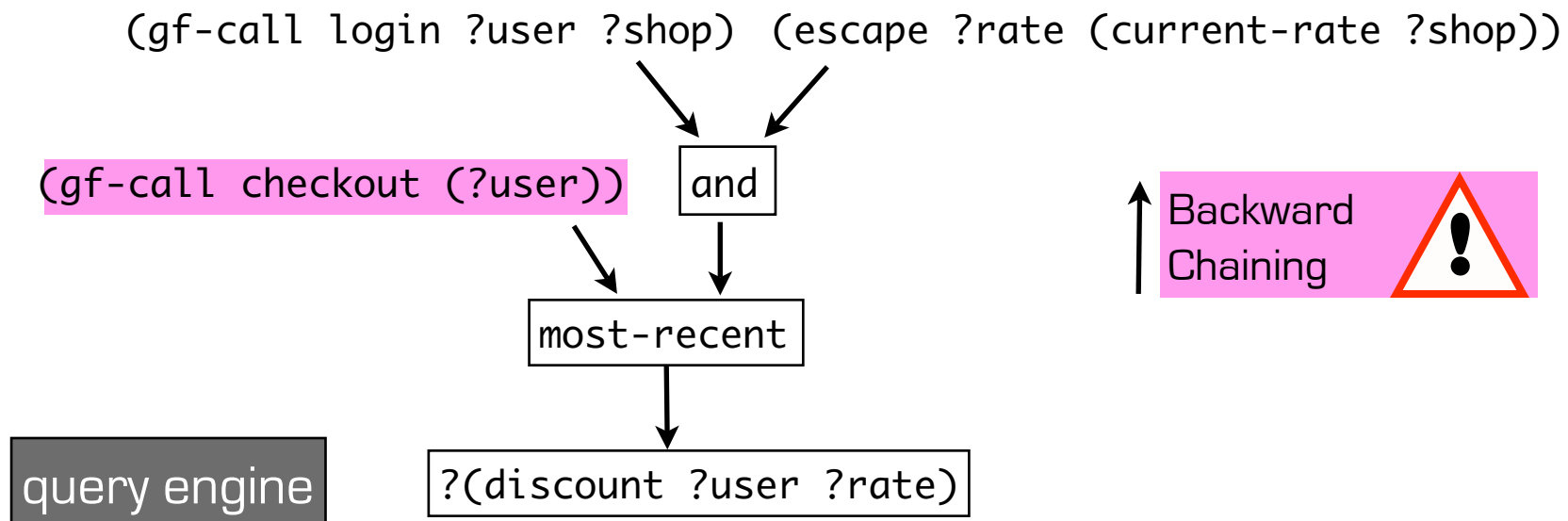
# Backward Chaining vs Rete Forward Chaining

```
(at ((gf-call checkout (?user))  
    (most-recent (gf-call login ?user ?shop)  
                  (escape ?rate (current-rate ?shop)))))  
(discount ?user ?rate))
```



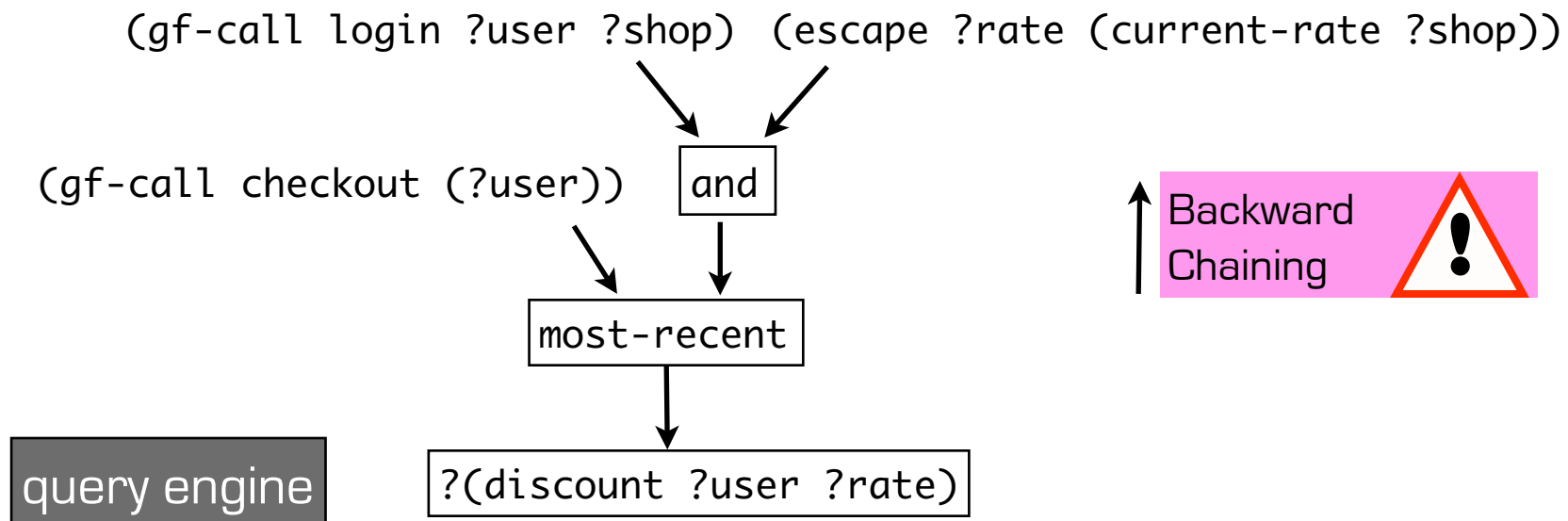
# Backward Chaining vs Rete Forward Chaining

```
(at ((gf-call checkout (?user))  
    (most-recent (gf-call login ?user ?shop)  
                  (escape ?rate (current-rate ?shop)))))  
(discount ?user ?rate))
```



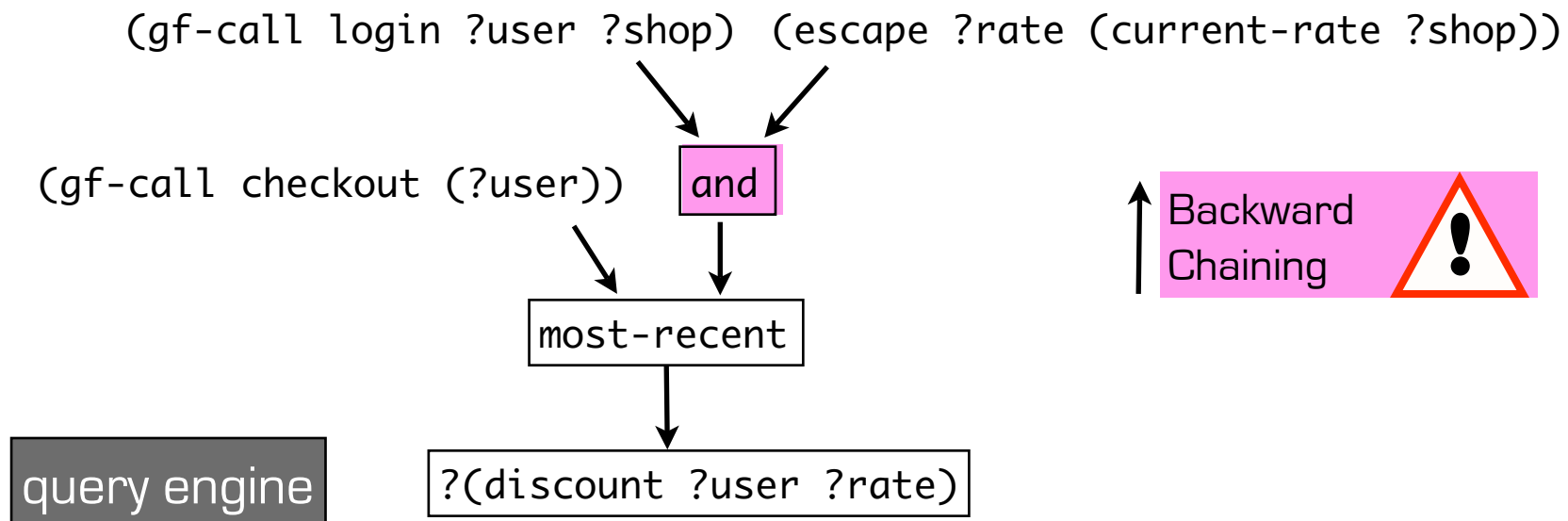
# Backward Chaining vs Rete Forward Chaining

```
(at ((gf-call checkout (?user))  
    (most-recent (gf-call login ?user ?shop)  
                  (escape ?rate (current-rate ?shop)))))  
(discount ?user ?rate))
```



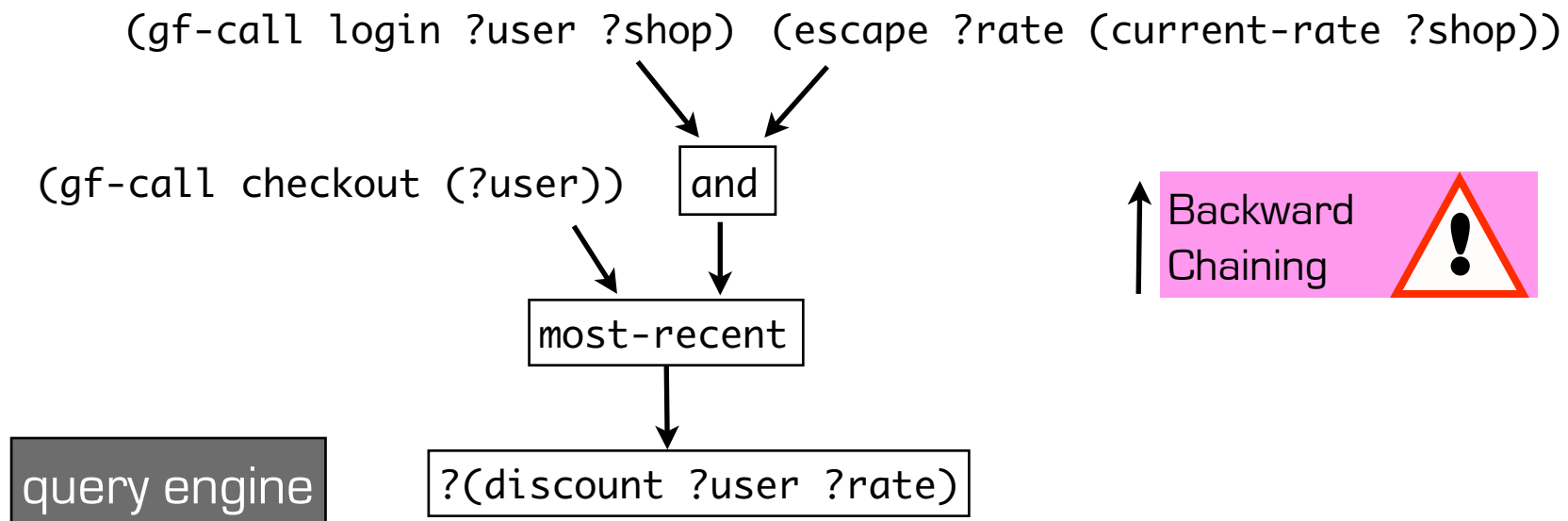
# Backward Chaining vs Rete Forward Chaining

```
(at ((gf-call checkout (?user))  
    (most-recent (gf-call login ?user ?shop)  
                  (escape ?rate (current-rate ?shop)))))  
(discount ?user ?rate))
```



# Backward Chaining vs Rete Forward Chaining

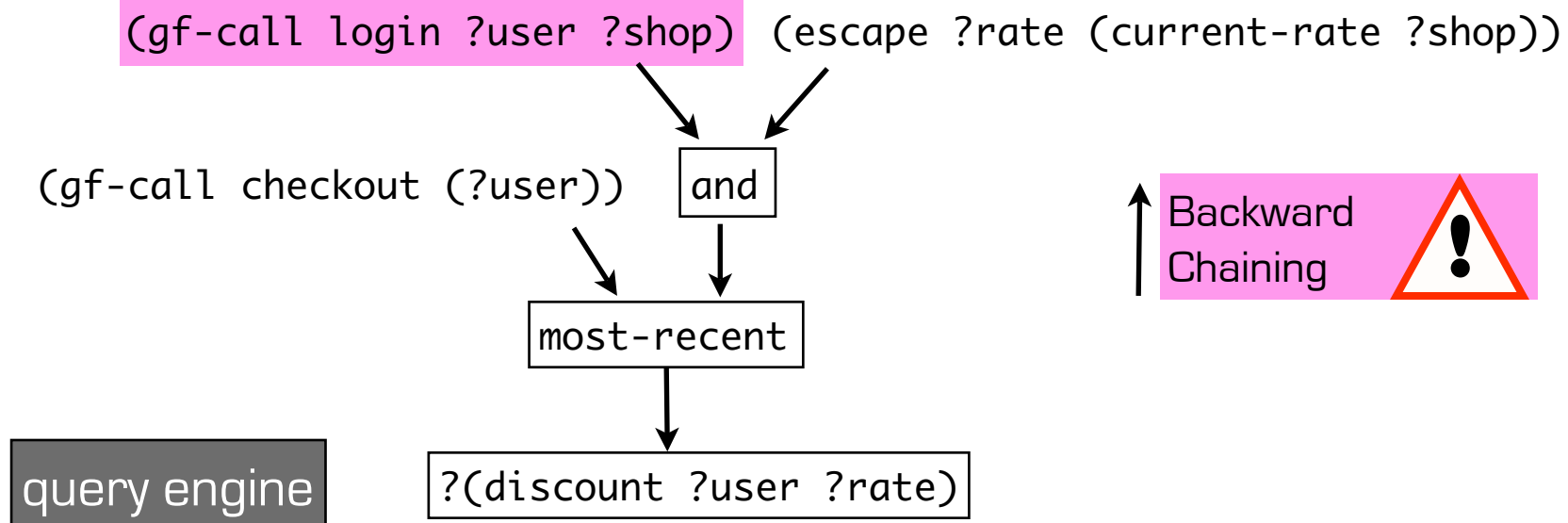
```
(at ((gf-call checkout (?user))  
    (most-recent (gf-call login ?user ?shop)  
                  (escape ?rate (current-rate ?shop)))))  
(discount ?user ?rate))
```





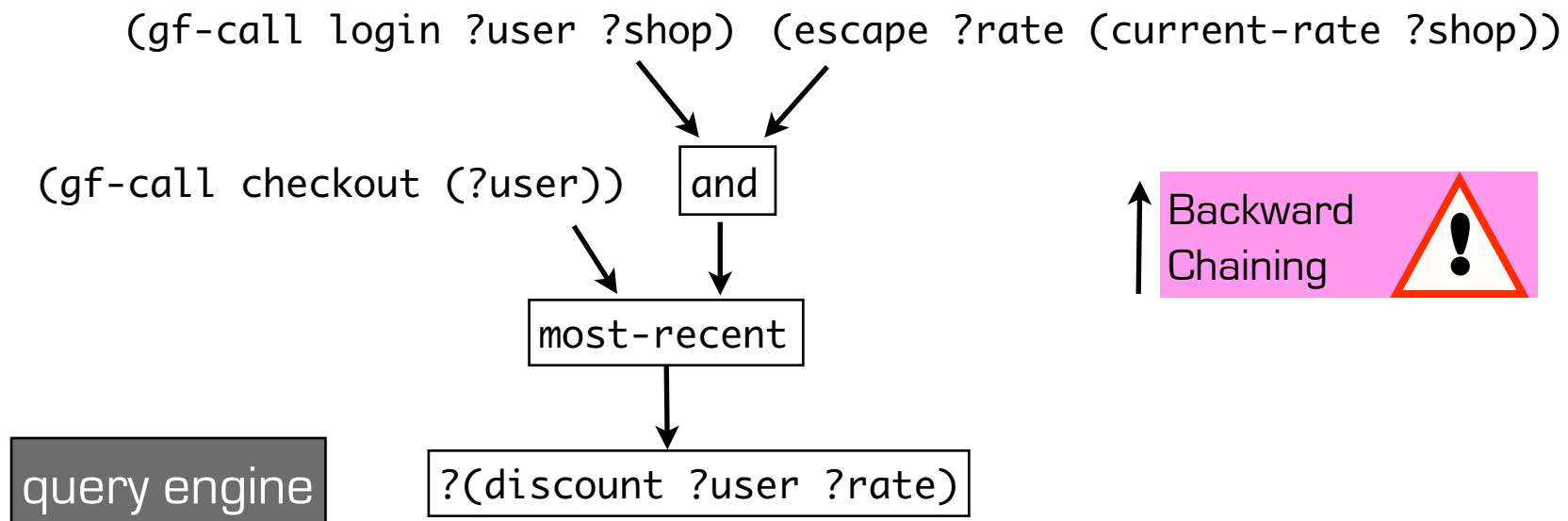
# Backward Chaining vs Rete Forward Chaining

```
(at ((gf-call checkout (?user))  
    (most-recent (gf-call login ?user ?shop)  
                  (escape ?rate (current-rate ?shop)))))  
(discount ?user ?rate))
```



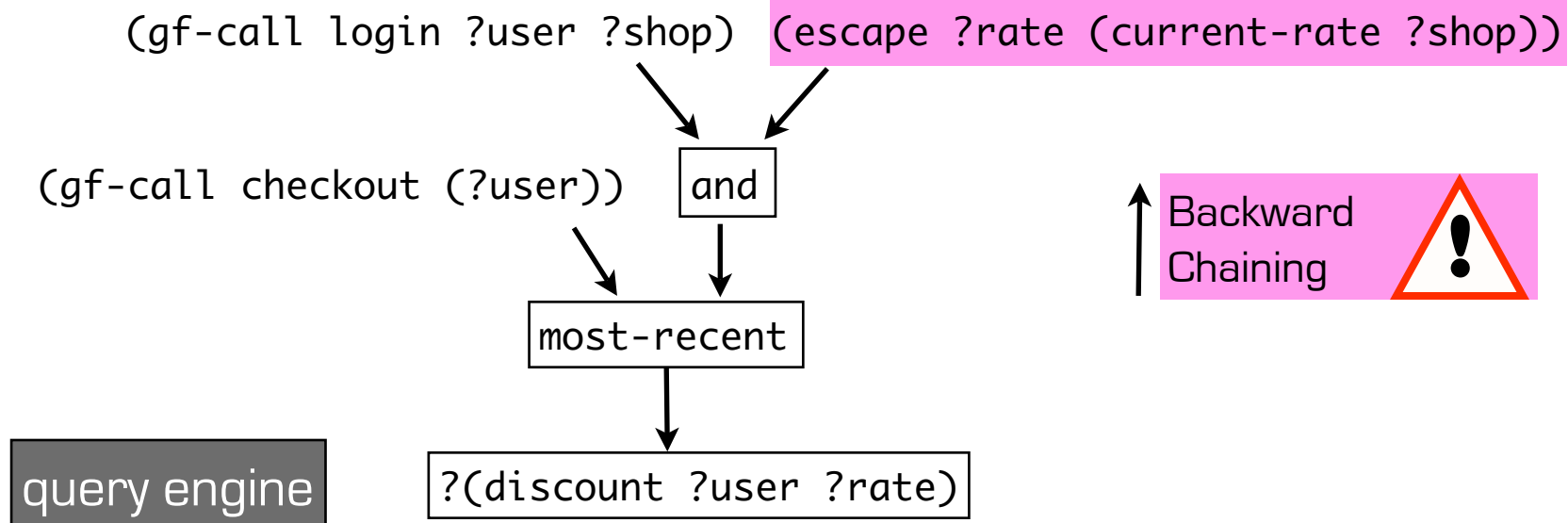
# Backward Chaining vs Rete Forward Chaining

```
(at ((gf-call checkout (?user))  
    (most-recent (gf-call login ?user ?shop)  
                  (escape ?rate (current-rate ?shop)))))  
(discount ?user ?rate))
```



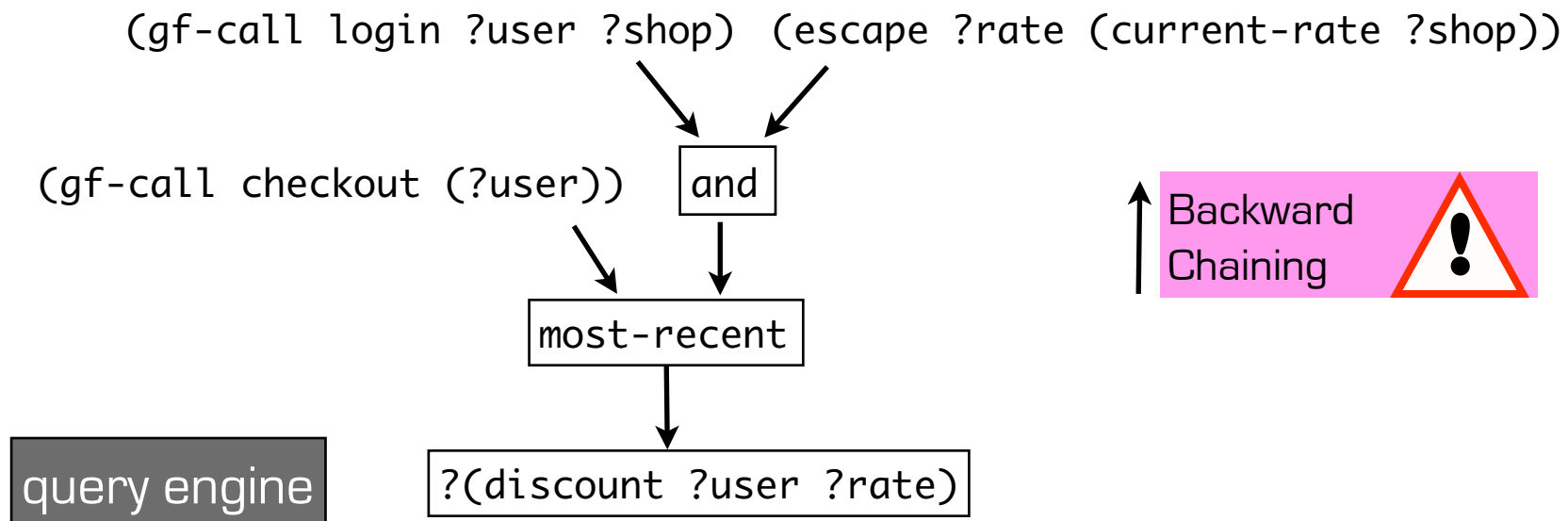
# Backward Chaining vs Rete Forward Chaining

```
(at ((gf-call checkout (?user))  
    (most-recent (gf-call login ?user ?shop)  
                  (escape ?rate (current-rate ?shop)))))  
(discount ?user ?rate))
```

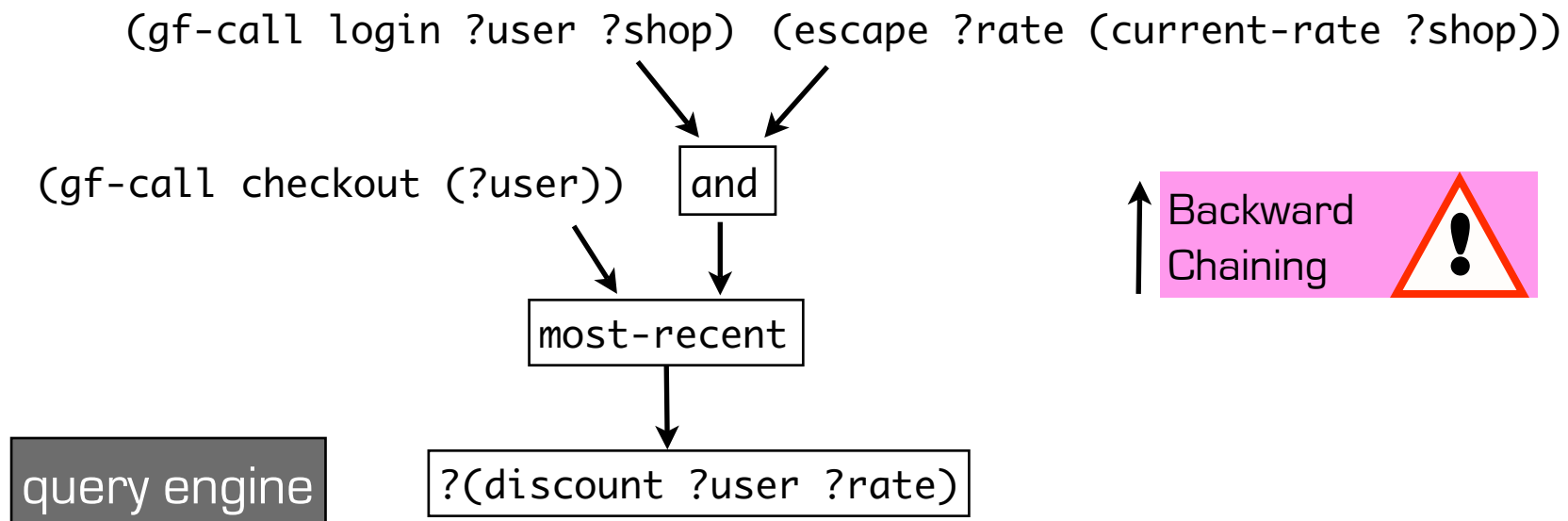


# Backward Chaining vs Rete Forward Chaining

```
(at ((gf-call checkout (?user))  
    (most-recent (gf-call login ?user ?shop)  
                  (escape ?rate (current-rate ?shop)))))  
(discount ?user ?rate))
```

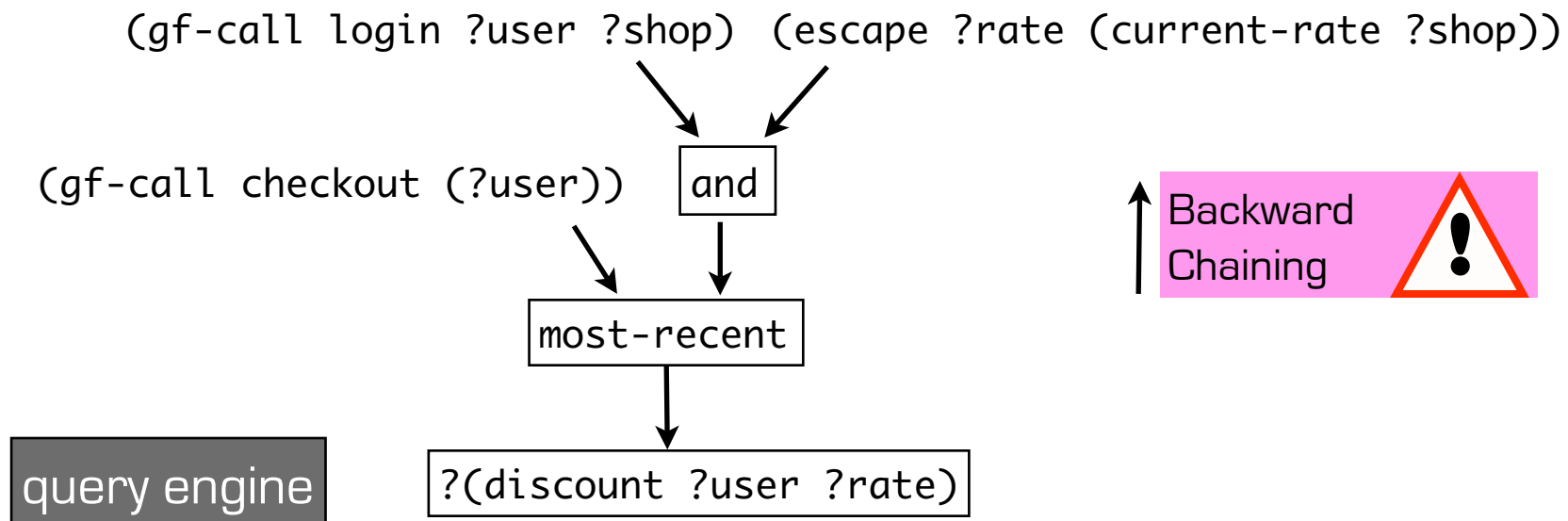


# Backward Chaining vs Rete Forward Chaining



# Backward Chaining vs Rete Forward Chaining

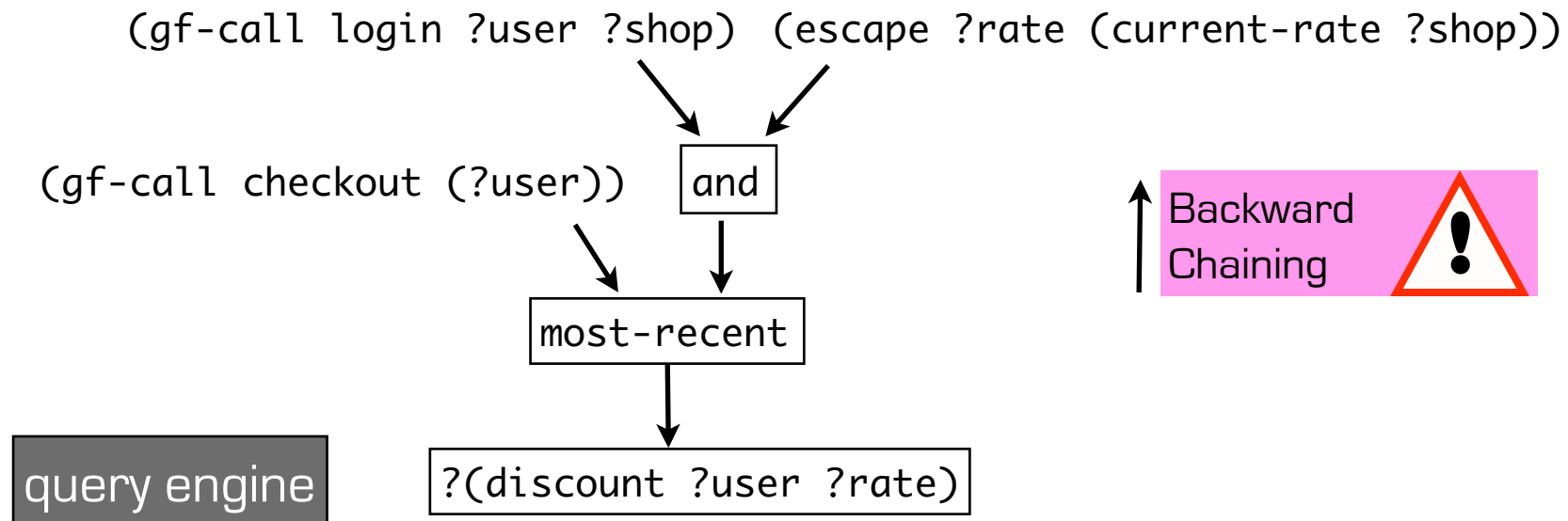
"On every checkout, give a discount  
based on the rate at login time"



# Backward Chaining vs Rete Forward Chaining

"On every checkout, give a discount  
based on the rate at login time"

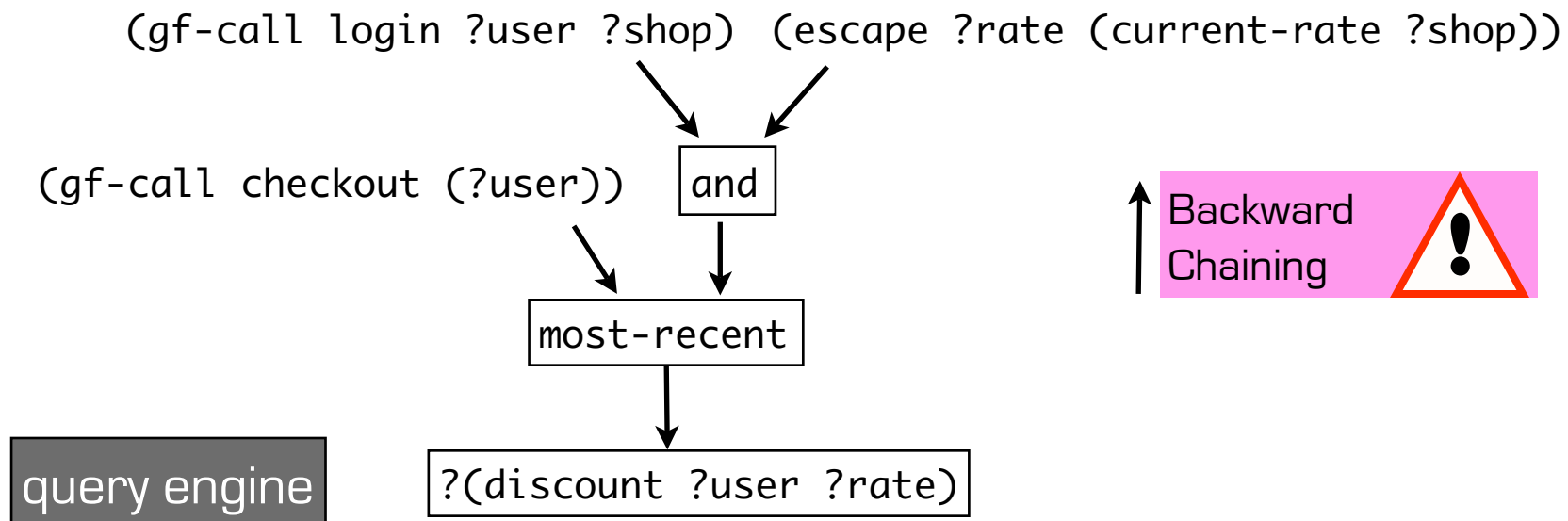
```
(set-rate *shop* 0.05)
(login *kris* *shop*)
(buy *kris* *cd*)
(set-rate *shop* 0.00)
(checkout *kris*)
```



# Backward Chaining vs Rete Forward Chaining

"On every checkout, give a discount based on the rate at login time"

```
1. (set-rate *shop* 0.05)
   (login *kris* *shop*)
   (buy *kris* *cd*)
   (set-rate *shop* 0.00)
   (checkout *kris*)
```



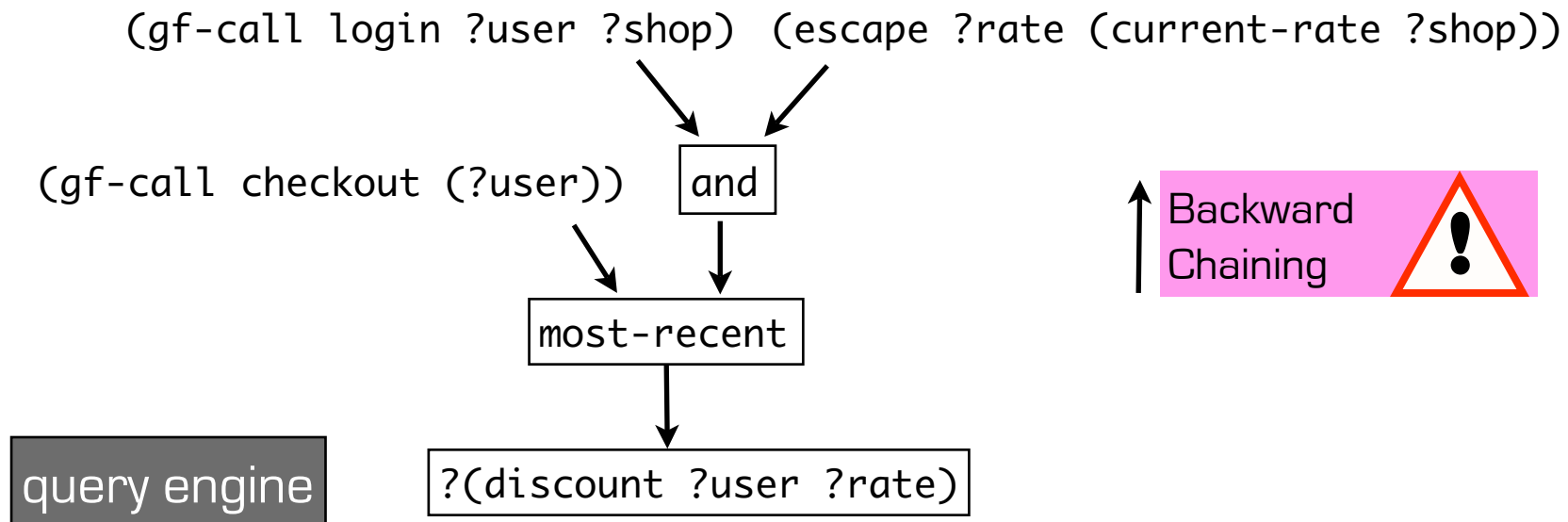


# Backward Chaining vs Rete Forward Chaining

"On every checkout, give a discount based on the rate at login time"

```
1. (set-rate *shop* 0.05)
   (login *kris* *shop*)
   (buy *kris* *cd*)
   (set-rate *shop* 0.00)
   (checkout *kris*)
```

```
(1 gf-call 'set-rate <shop> 0.05)
```

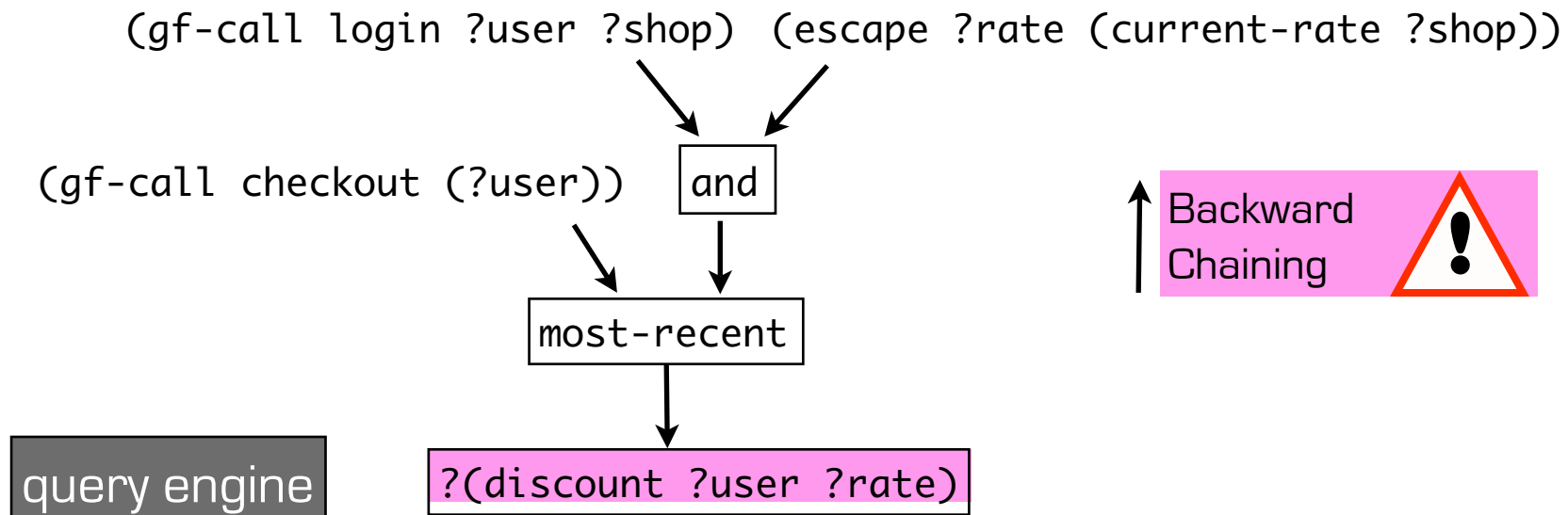


# Backward Chaining vs Rete Forward Chaining

"On every checkout, give a discount based on the rate at login time"

```
1. (set-rate *shop* 0.05)
   (login *kris* *shop*)
   (buy *kris* *cd*)
   (set-rate *shop* 0.00)
   (checkout *kris*)
```

```
(1 gf-call 'set-rate <shop> 0.05)
```

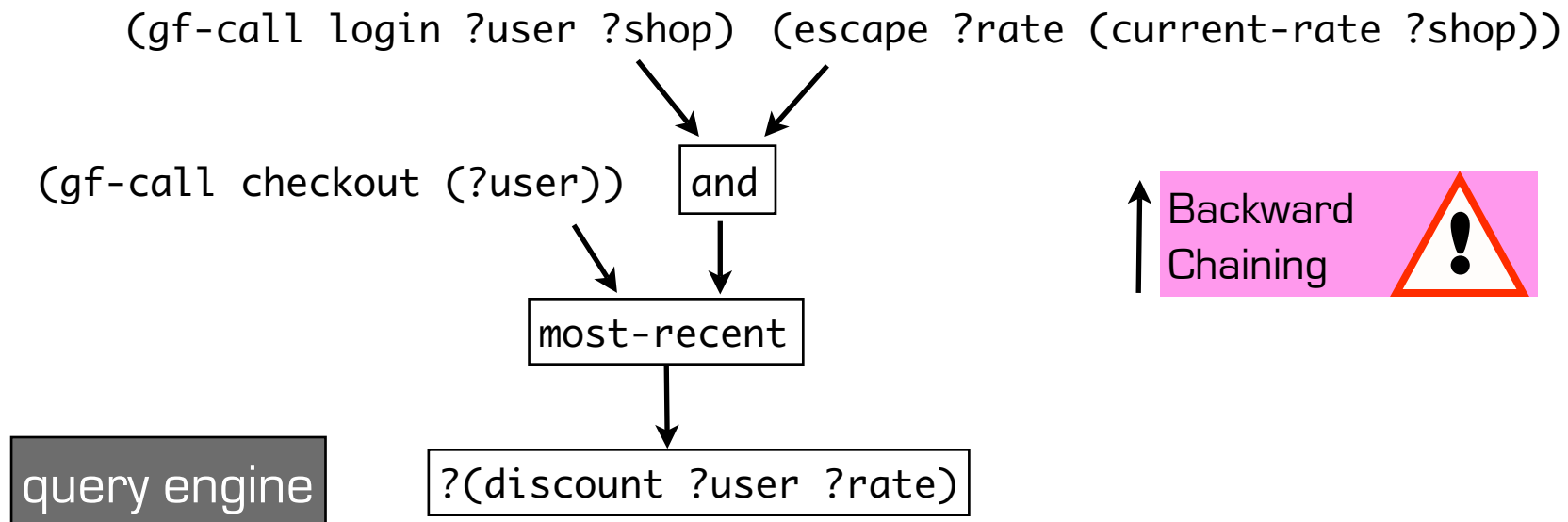


# Backward Chaining vs Rete Forward Chaining

"On every checkout, give a discount based on the rate at login time"

```
1. (set-rate *shop* 0.05)
   (login *kris* *shop*)
   (buy *kris* *cd*)
   (set-rate *shop* 0.00)
   (checkout *kris*)
```

```
(1 gf-call 'set-rate <shop> 0.05)
```

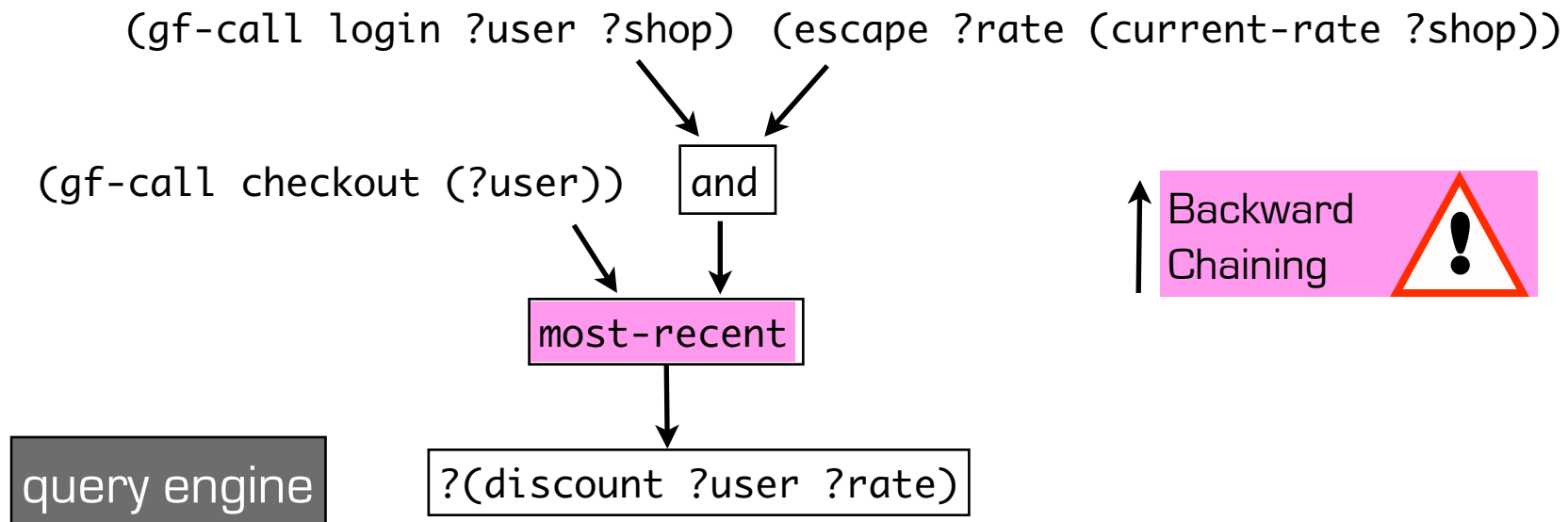


# Backward Chaining vs Rete Forward Chaining

"On every checkout, give a discount based on the rate at login time"

```
1. (set-rate *shop* 0.05)
   (login *kris* *shop*)
   (buy *kris* *cd*)
   (set-rate *shop* 0.00)
   (checkout *kris*)
```

```
(1 gf-call 'set-rate <shop> 0.05)
```

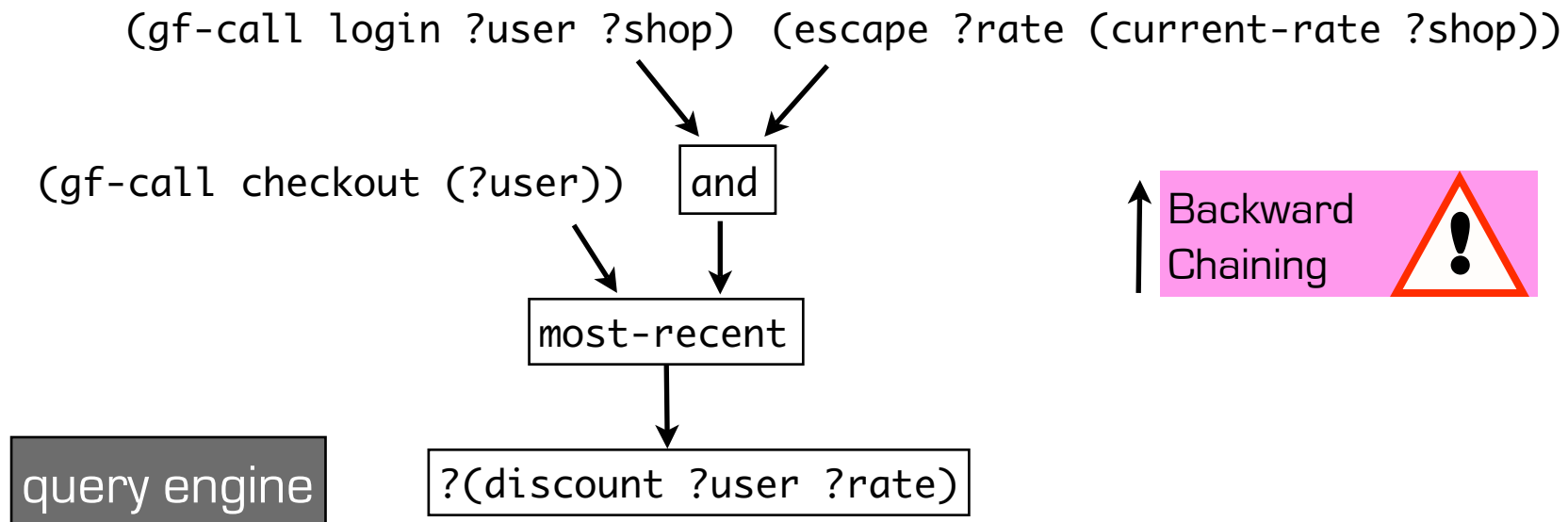


# Backward Chaining vs Rete Forward Chaining

"On every checkout, give a discount based on the rate at login time"

```
1. (set-rate *shop* 0.05)
   (login *kris* *shop*)
   (buy *kris* *cd*)
   (set-rate *shop* 0.00)
   (checkout *kris*)
```

```
(1 gf-call 'set-rate <shop> 0.05)
```



# Backward Chaining vs Rete Forward Chaining

"On every checkout, give a discount based on the rate at login time"

```
1. (set-rate *shop* 0.05)
   (login *kris* *shop*)
   (buy *kris* *cd*)
   (set-rate *shop* 0.00)
   (checkout *kris*)
```

```
(1 gf-call 'set-rate <shop> 0.05)
```

```
(gf-call login ?user ?shop) (escape ?rate (current-rate ?shop))
```


```
(gf-call checkout (?user))
```

and

most-recent

query engine

?(discount ?user ?rate)

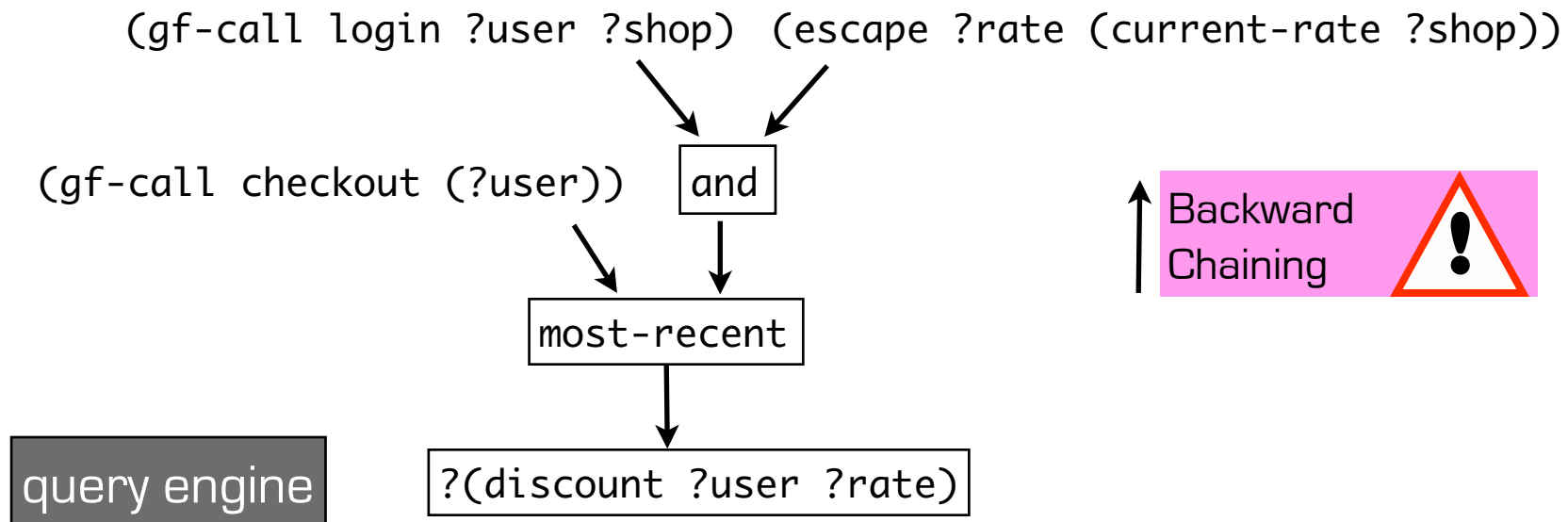
↑ Backward Chaining 

# Backward Chaining vs Rete Forward Chaining

"On every checkout, give a discount based on the rate at login time"

```
1. (set-rate *shop* 0.05)
   (login *kris* *shop*)
   (buy *kris* *cd*)
   (set-rate *shop* 0.00)
   (checkout *kris*)
```

```
(1 gf-call 'set-rate <shop> 0.05)
```

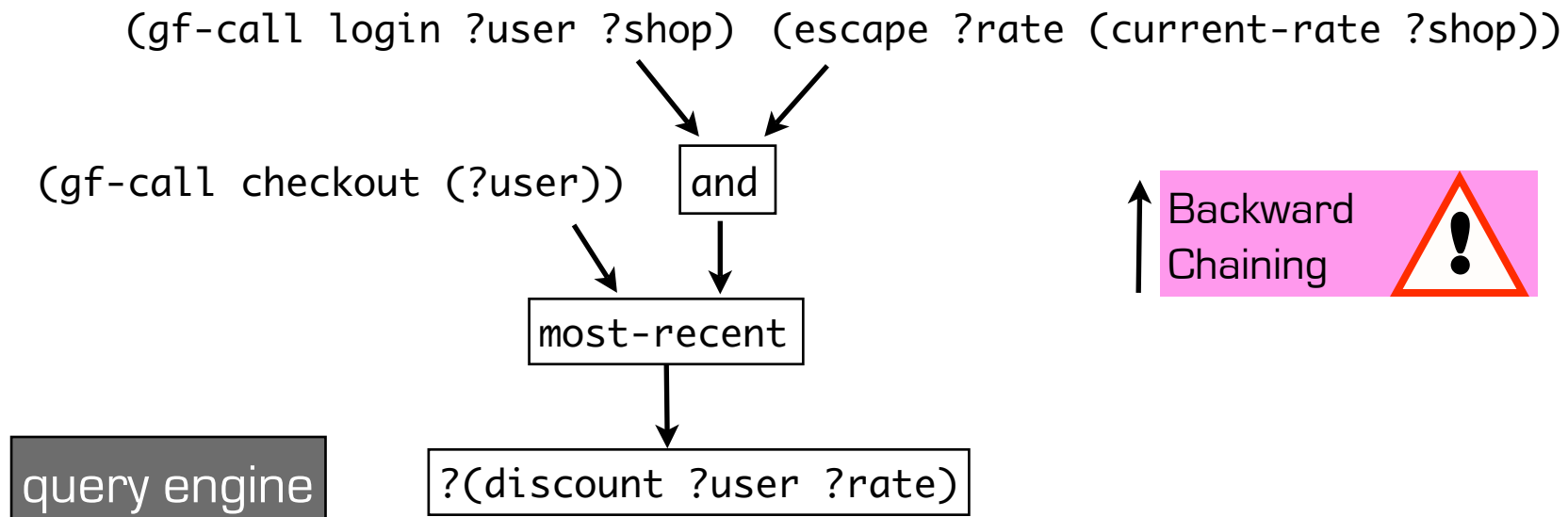


# Backward Chaining vs Rete Forward Chaining

"On every checkout, give a discount based on the rate at login time"

```
1. (set-rate *shop* 0.05)
2. (login *kris* *shop*)
   (buy *kris* *cd*)
   (set-rate *shop* 0.00)
   (checkout *kris*)
```

```
(1 gf-call 'set-rate <shop> 0.05)
```



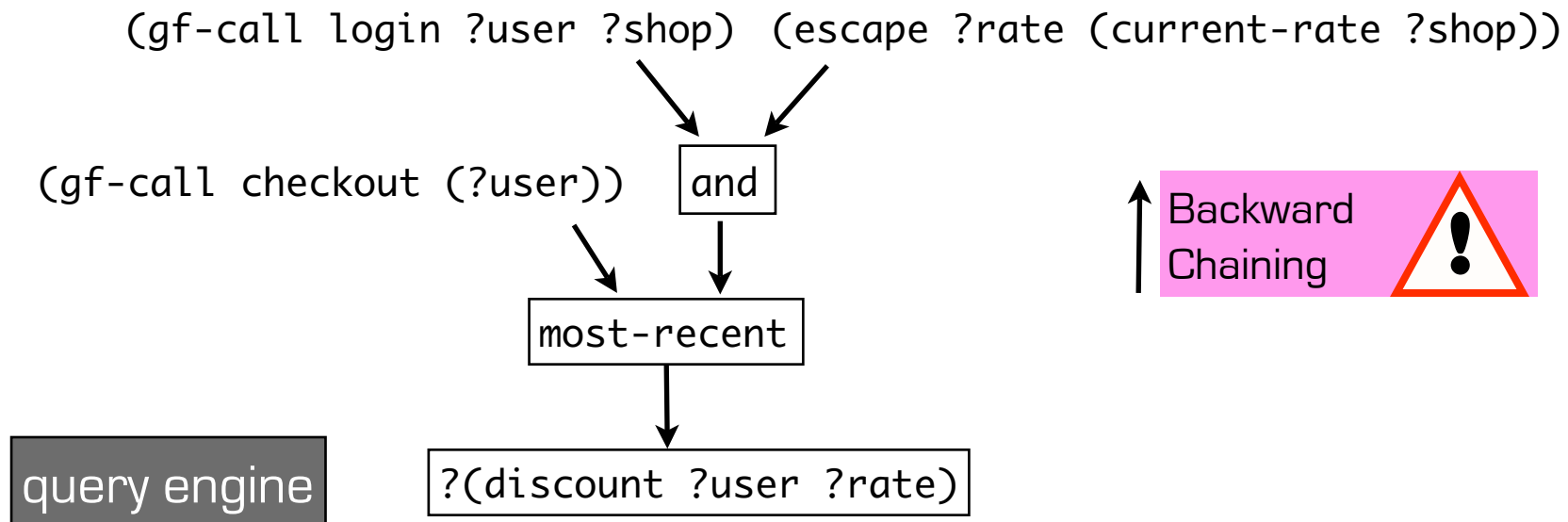


# Backward Chaining vs Rete Forward Chaining

"On every checkout, give a discount based on the rate at login time"

```
1. (set-rate *shop* 0.05)
2. (login *kris* *shop*)
   (buy *kris* *cd*)
   (set-rate *shop* 0.00)
   (checkout *kris*)
```

```
(1 gf-call 'set-rate <shop> 0.05)
(2 gf-call 'login <kris> <shop>)
```

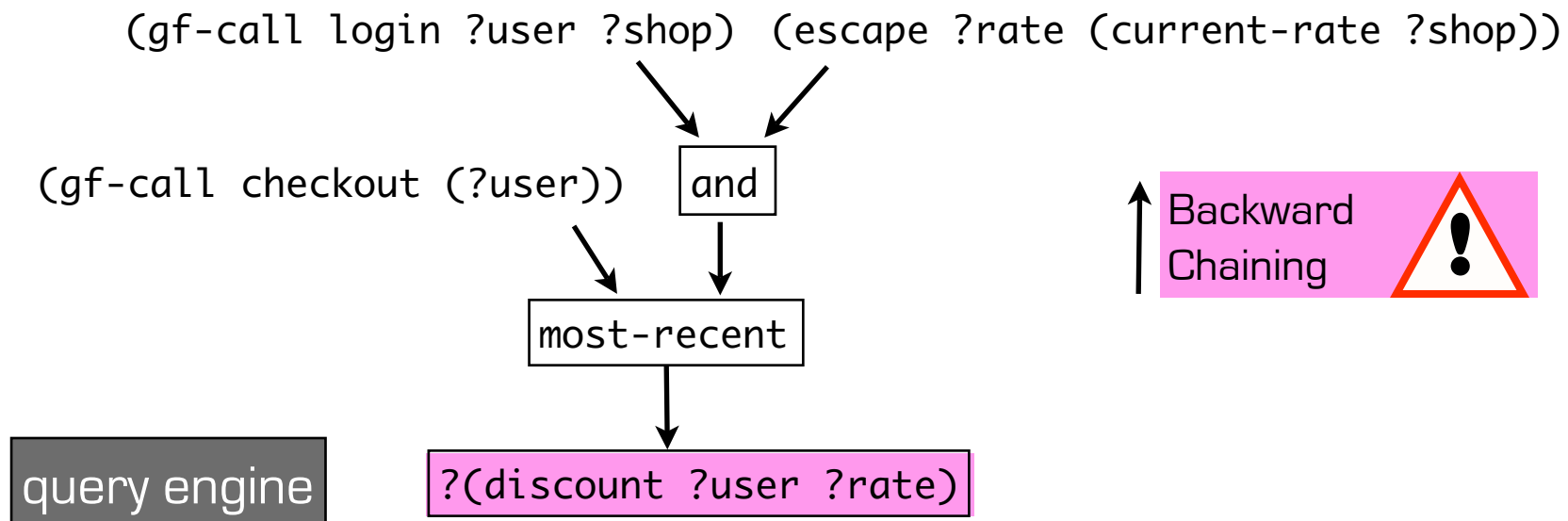


# Backward Chaining vs Rete Forward Chaining

"On every checkout, give a discount based on the rate at login time"

```
1. (set-rate *shop* 0.05)
2. (login *kris* *shop*)
   (buy *kris* *cd*)
   (set-rate *shop* 0.00)
   (checkout *kris*)
```

```
(1 gf-call 'set-rate <shop> 0.05)
(2 gf-call 'login <kris> <shop>)
```

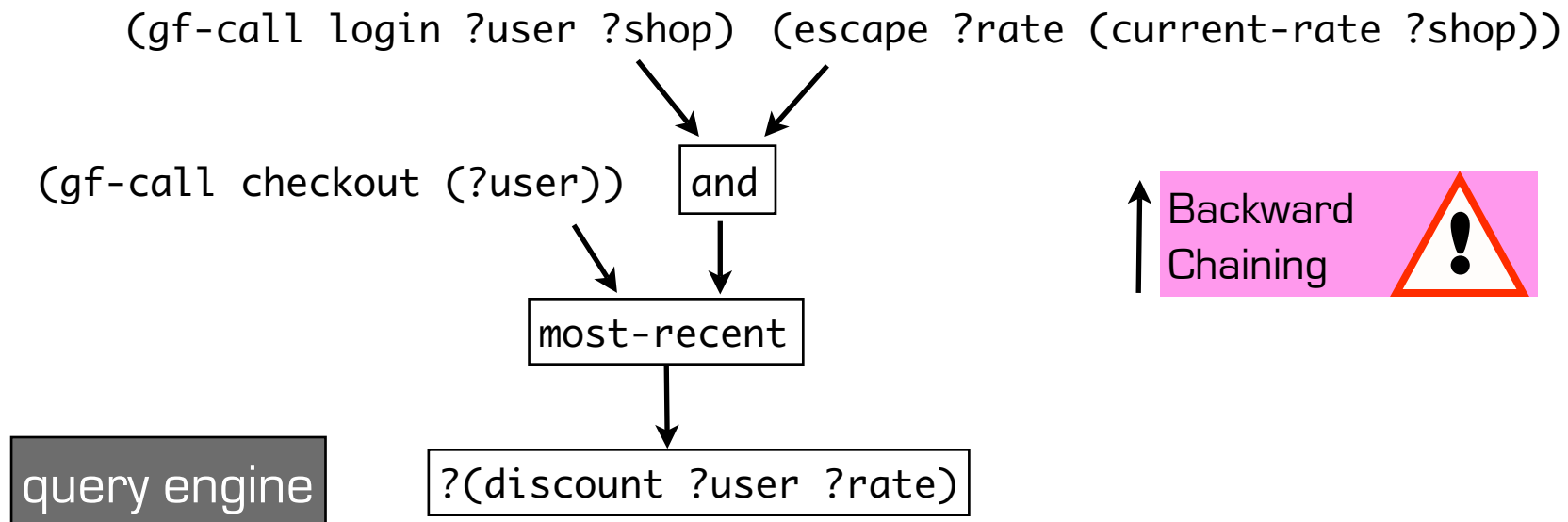


# Backward Chaining vs Rete Forward Chaining

"On every checkout, give a discount based on the rate at login time"

```
1. (set-rate *shop* 0.05)
2. (login *kris* *shop*)
   (buy *kris* *cd*)
   (set-rate *shop* 0.00)
   (checkout *kris*)
```

```
(1 gf-call 'set-rate <shop> 0.05)
(2 gf-call 'login <kris> <shop>)
```

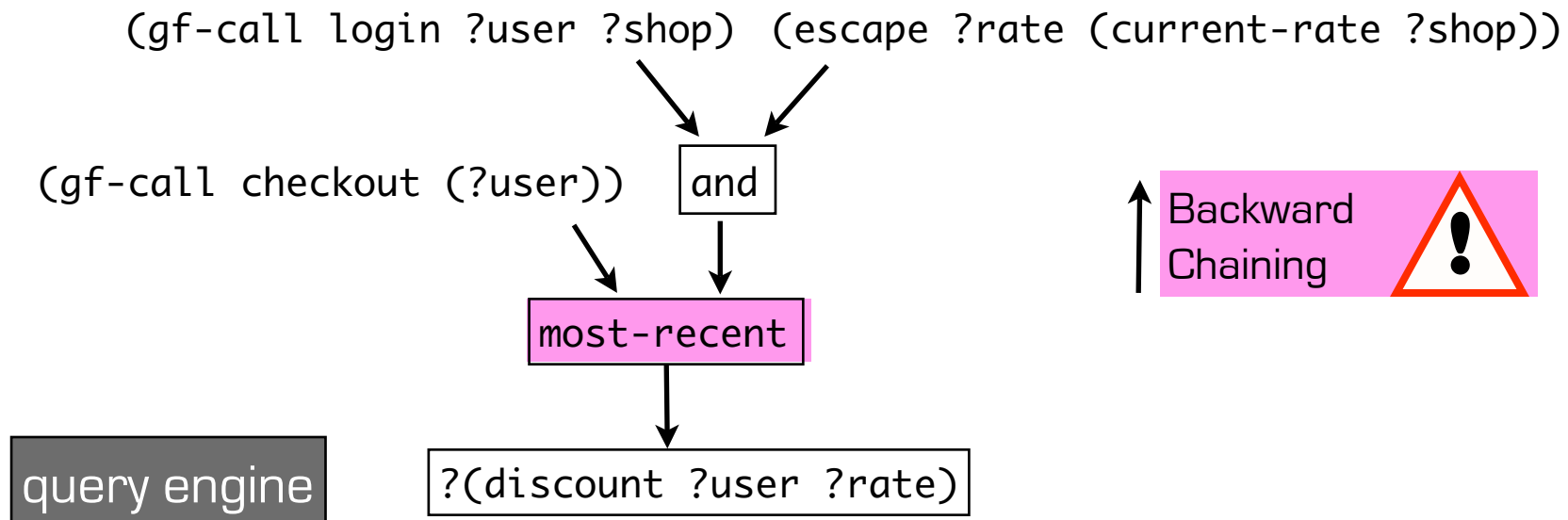


# Backward Chaining vs Rete Forward Chaining

"On every checkout, give a discount based on the rate at login time"

```
1. (set-rate *shop* 0.05)
2. (login *kris* *shop*)
   (buy *kris* *cd*)
   (set-rate *shop* 0.00)
   (checkout *kris*)
```

```
(1 gf-call 'set-rate <shop> 0.05)
(2 gf-call 'login <kris> <shop>)
```

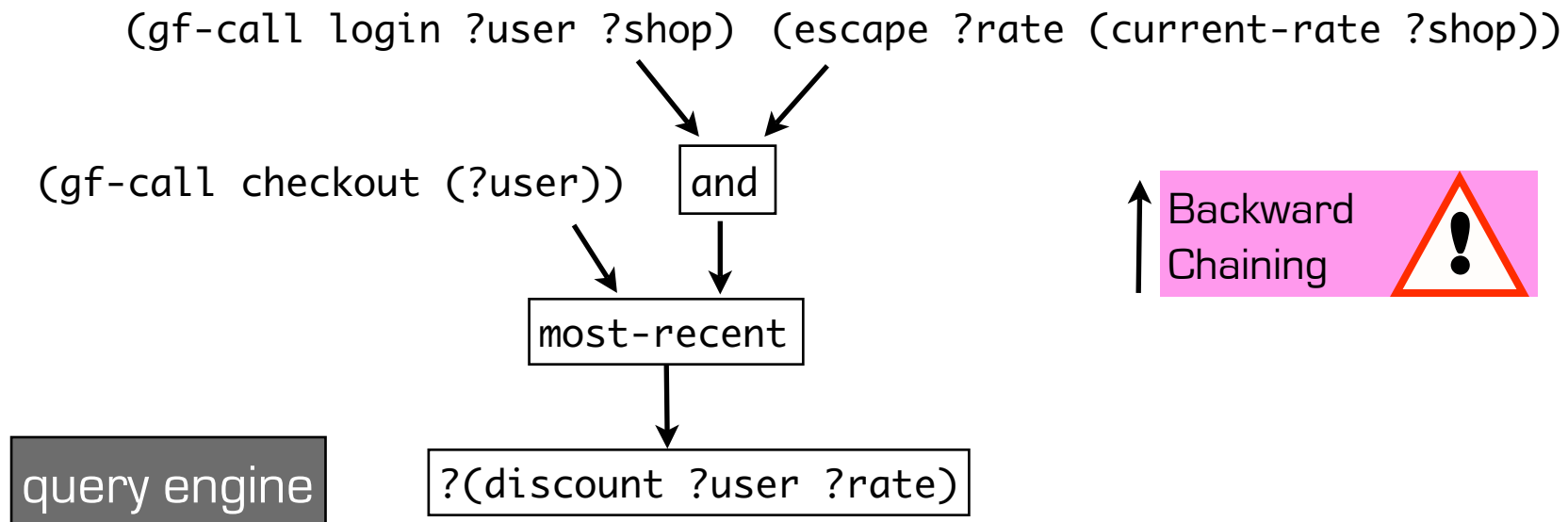


# Backward Chaining vs Rete Forward Chaining

"On every checkout, give a discount based on the rate at login time"

```
1. (set-rate *shop* 0.05)
2. (login *kris* *shop*)
   (buy *kris* *cd*)
   (set-rate *shop* 0.00)
   (checkout *kris*)
```

```
(1 gf-call 'set-rate <shop> 0.05)
(2 gf-call 'login <kris> <shop>)
```



# Backward Chaining vs Rete Forward Chaining

"On every checkout, give a discount based on the rate at login time"

```
1. (set-rate *shop* 0.05)
2. (login *kris* *shop*)
   (buy *kris* *cd*)
   (set-rate *shop* 0.00)
   (checkout *kris*)
```

```
(1 gf-call 'set-rate <shop> 0.05)
(2 gf-call 'login <kris> <shop>)
```

```
(gf-call login ?user ?shop) (escape ?rate (current-rate ?shop))
```


```
(gf-call checkout (?user))
```

and

most-recent

query engine

?(discount ?user ?rate)

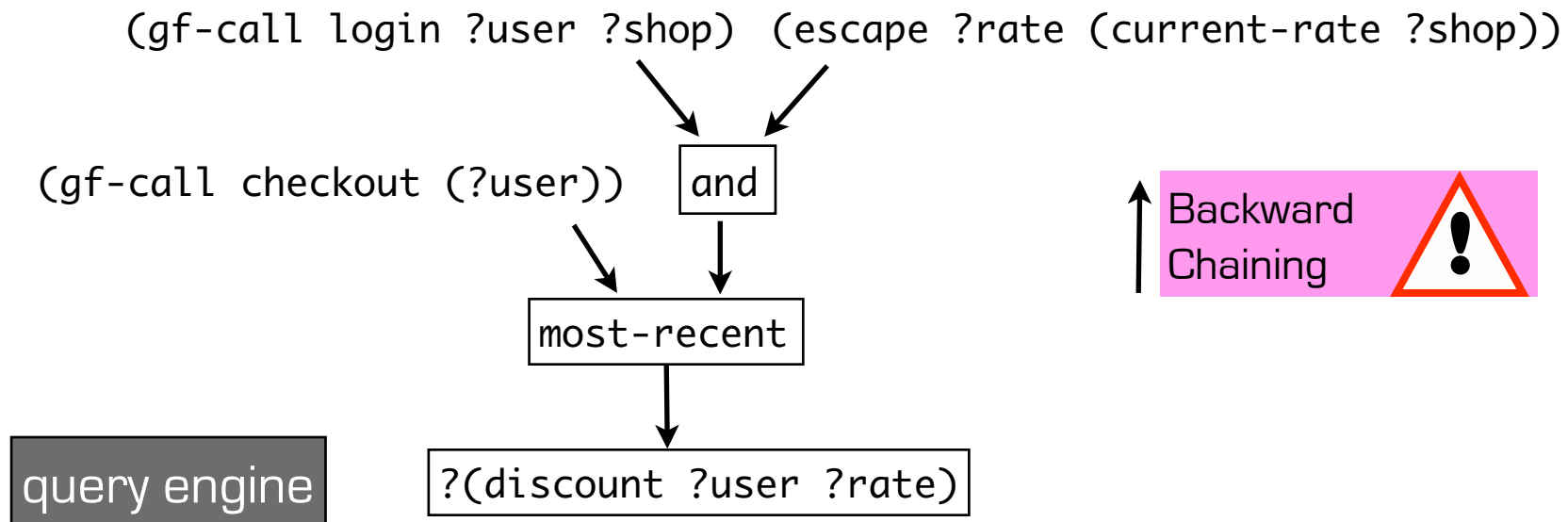
↑ Backward Chaining 

# Backward Chaining vs Rete Forward Chaining

"On every checkout, give a discount based on the rate at login time"

```
1. (set-rate *shop* 0.05)
2. (login *kris* *shop*)
   (buy *kris* *cd*)
   (set-rate *shop* 0.00)
   (checkout *kris*)
```

```
(1 gf-call 'set-rate <shop> 0.05)
(2 gf-call 'login <kris> <shop>)
```

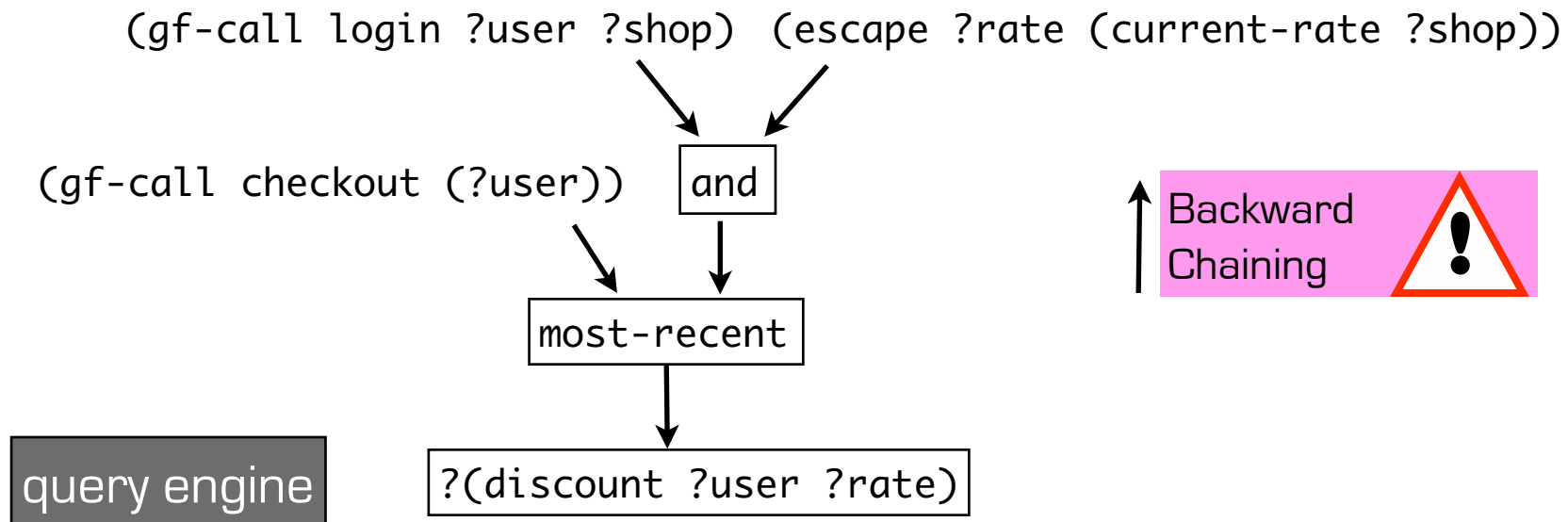


# Backward Chaining vs Rete Forward Chaining

"On every checkout, give a discount based on the rate at login time"

```
1. (set-rate *shop* 0.05)
2. (login *kris* *shop*)
3. (buy *kris* *cd*)
   (set-rate *shop* 0.00)
   (checkout *kris*)
```

```
(1 gf-call 'set-rate <shop> 0.05)
(2 gf-call 'login <kris> <shop>)
```



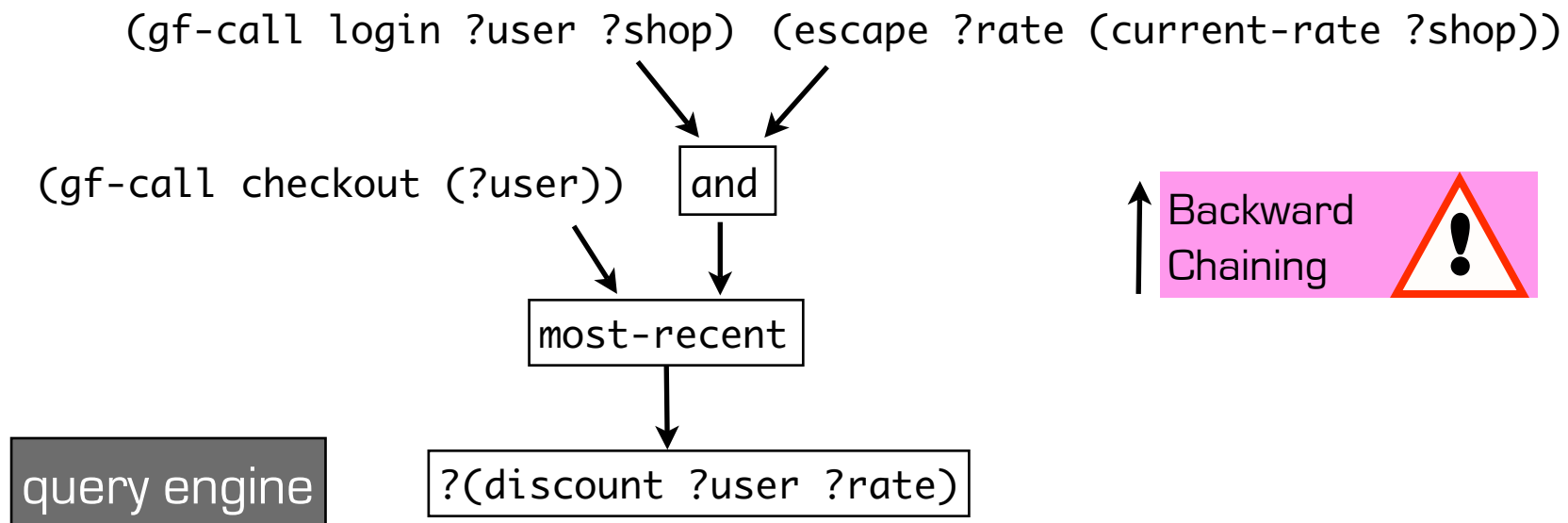


# Backward Chaining vs Rete Forward Chaining

"On every checkout, give a discount based on the rate at login time"

```
1. (set-rate *shop* 0.05)
2. (login *kris* *shop*)
3. (buy *kris* *cd*)
   (set-rate *shop* 0.00)
   (checkout *kris*)
```

```
(1 gf-call 'set-rate <shop> 0.05)
(2 gf-call 'login <kris> <shop>)
(3 gf-call 'buy <kris> <cd>)
```

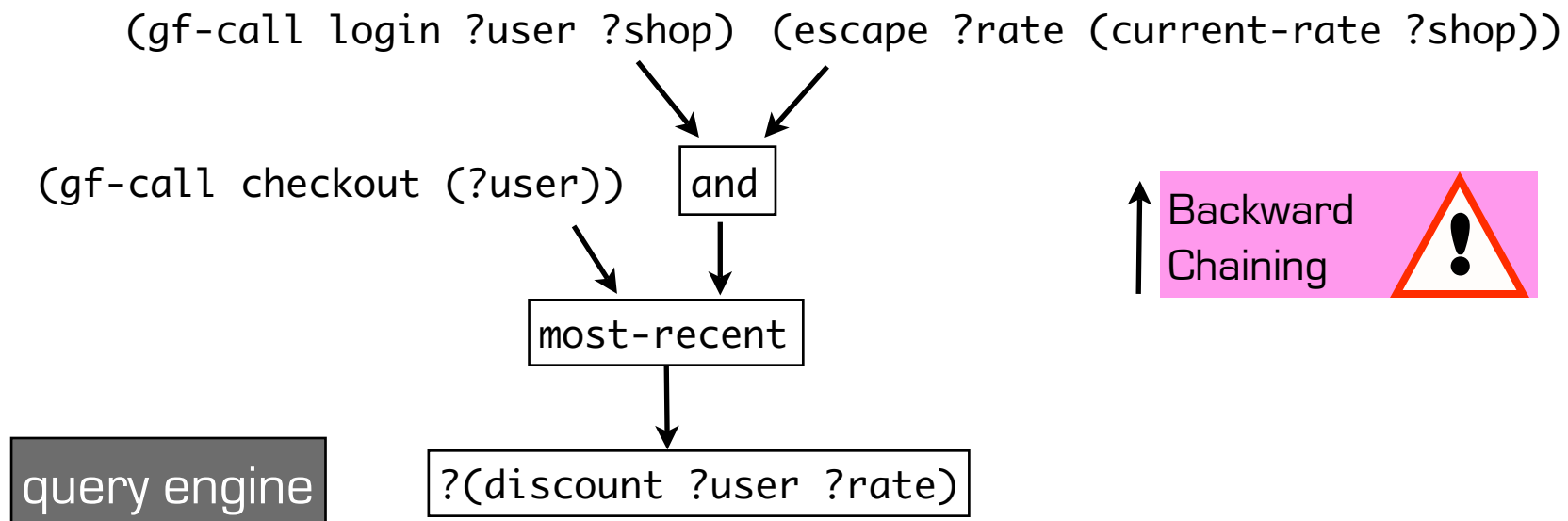


# Backward Chaining vs Rete Forward Chaining

"On every checkout, give a discount based on the rate at login time"

```
1. (set-rate *shop* 0.05)
2. (login *kris* *shop*)
3. (buy *kris* *cd*)
4. (set-rate *shop* 0.00)
   (checkout *kris*)
```

```
(1 gf-call 'set-rate <shop> 0.05)
(2 gf-call 'login <kris> <shop>)
(3 gf-call 'buy <kris> <cd>)
```

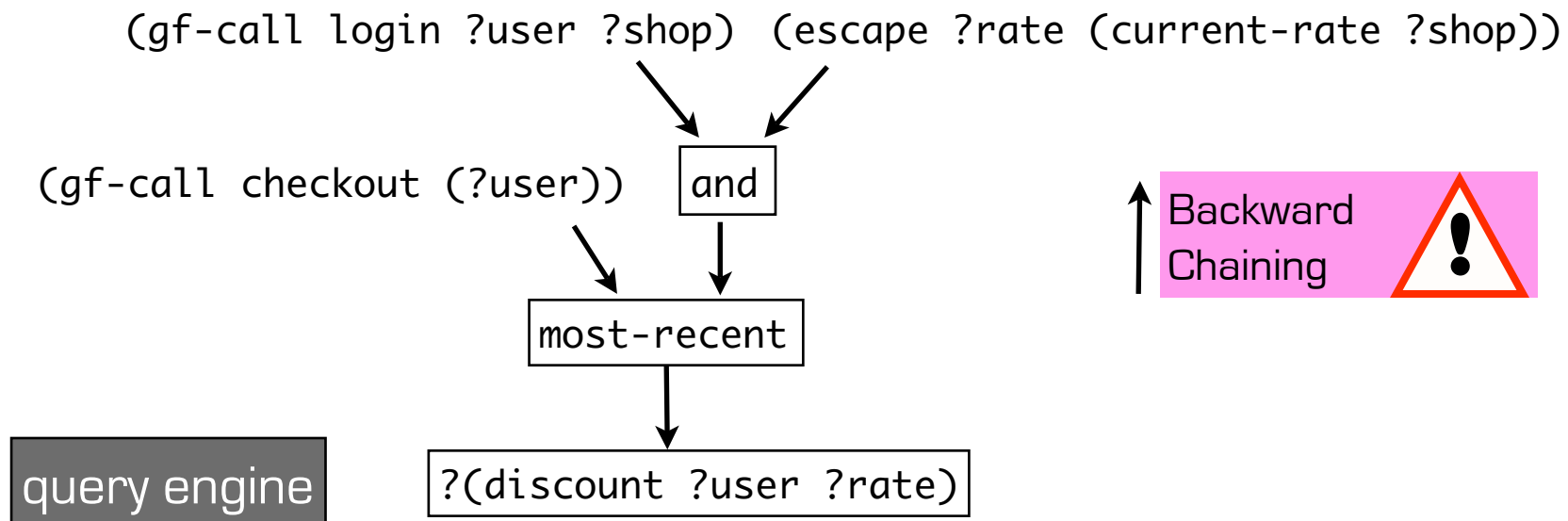


# Backward Chaining vs Rete Forward Chaining

"On every checkout, give a discount based on the rate at login time"

```
1. (set-rate *shop* 0.05)
2. (login *kris* *shop*)
3. (buy *kris* *cd*)
4. (set-rate *shop* 0.00)
   (checkout *kris*)
```

```
(1 gf-call 'set-rate <shop> 0.05)
(2 gf-call 'login <kris> <shop>)
(3 gf-call 'buy <kris> <cd>)
(4 gf-call 'set-rate <shop> 0.00)
```

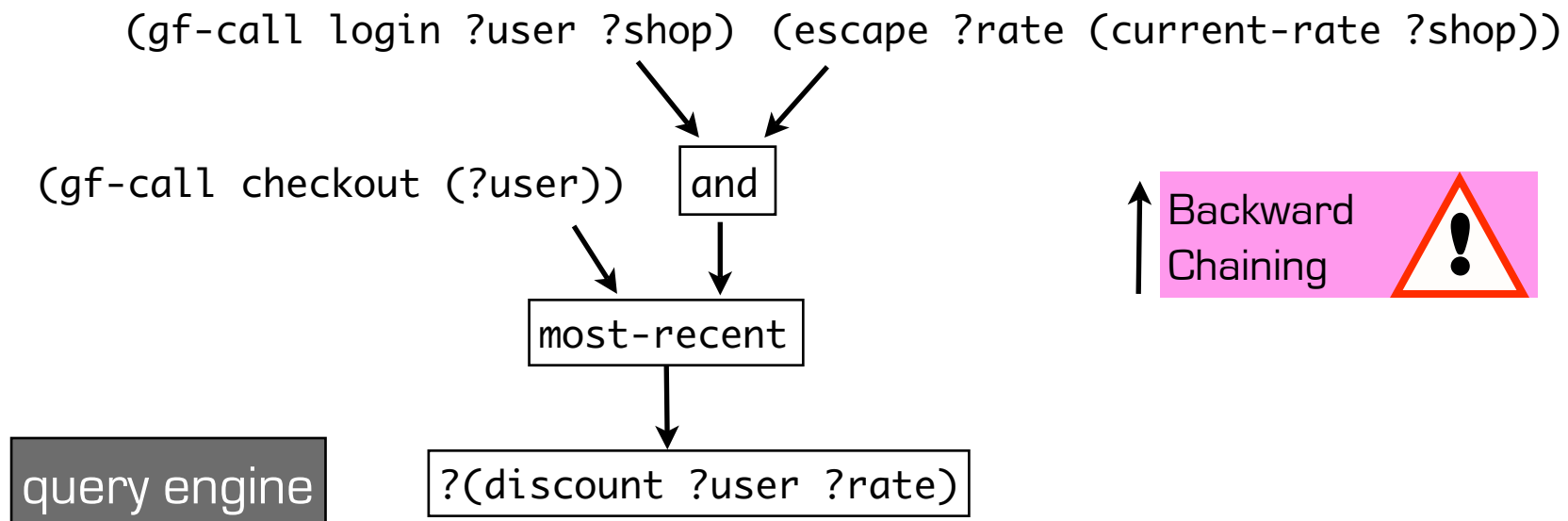


# Backward Chaining vs Rete Forward Chaining

"On every checkout, give a discount based on the rate at login time"

```
1. (set-rate *shop* 0.05)
2. (login *kris* *shop*)
3. (buy *kris* *cd*)
4. (set-rate *shop* 0.00)
5. (checkout *kris*)
```

```
(1 gf-call 'set-rate <shop> 0.05)
(2 gf-call 'login <kris> <shop>)
(3 gf-call 'buy <kris> <cd>)
(4 gf-call 'set-rate <shop> 0.00)
```

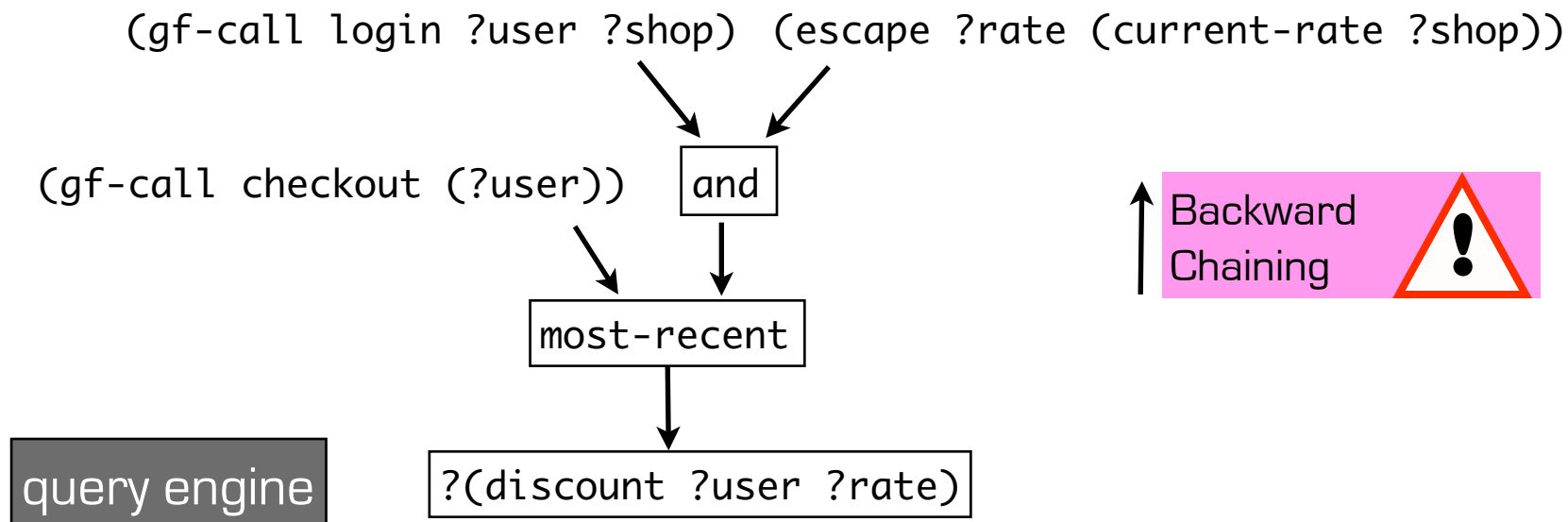


# Backward Chaining vs Rete Forward Chaining

"On every checkout, give a discount based on the rate at login time"

```
1. (set-rate *shop* 0.05)
2. (login *kris* *shop*)
3. (buy *kris* *cd*)
4. (set-rate *shop* 0.00)
5. (checkout *kris*)
```

```
(1 gf-call 'set-rate <shop> 0.05)
(2 gf-call 'login <kris> <shop>)
(3 gf-call 'buy <kris> <cd>)
(4 gf-call 'set-rate <shop> 0.00)
(5 gf-call 'checkout <kris>)
```

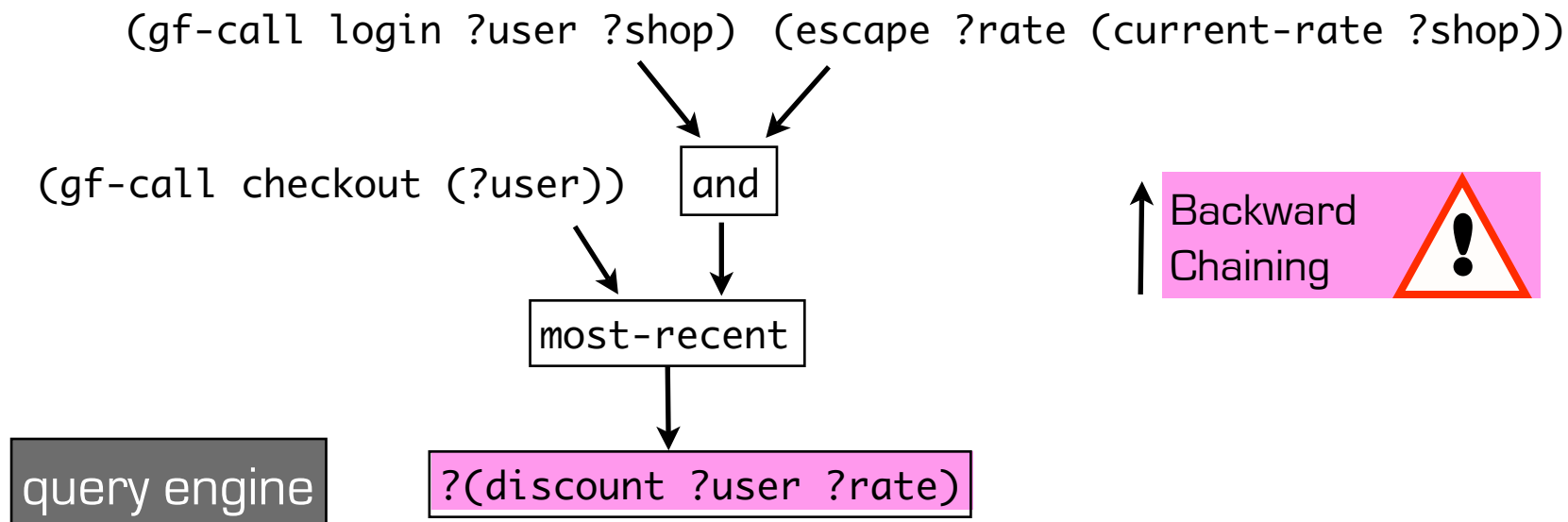


# Backward Chaining vs Rete Forward Chaining

"On every checkout, give a discount based on the rate at login time"

```
1. (set-rate *shop* 0.05)
2. (login *kris* *shop*)
3. (buy *kris* *cd*)
4. (set-rate *shop* 0.00)
5. (checkout *kris*)
```

```
(1 gf-call 'set-rate <shop> 0.05)
(2 gf-call 'login <kris> <shop>)
(3 gf-call 'buy <kris> <cd>)
(4 gf-call 'set-rate <shop> 0.00)
(5 gf-call 'checkout <kris>)
```

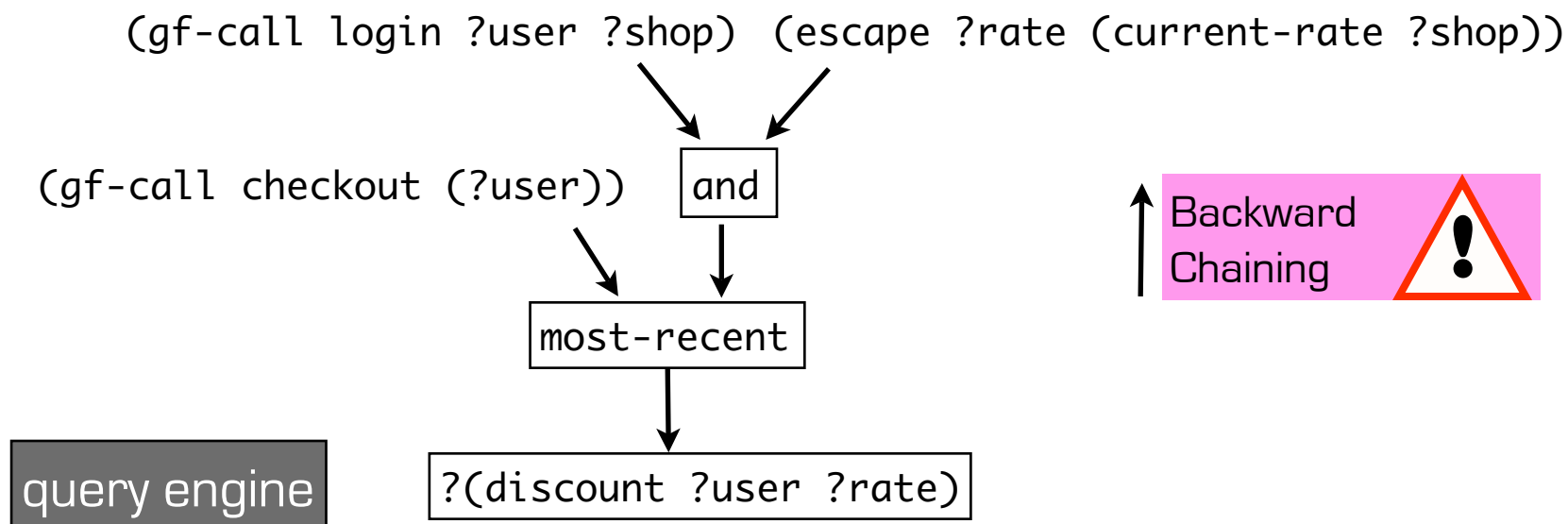


# Backward Chaining vs Rete Forward Chaining

"On every checkout, give a discount based on the rate at login time"

```
1. (set-rate *shop* 0.05)
2. (login *kris* *shop*)
3. (buy *kris* *cd*)
4. (set-rate *shop* 0.00)
5. (checkout *kris*)
```

```
(1 gf-call 'set-rate <shop> 0.05)
(2 gf-call 'login <kris> <shop>)
(3 gf-call 'buy <kris> <cd>)
(4 gf-call 'set-rate <shop> 0.00)
(5 gf-call 'checkout <kris>)
```

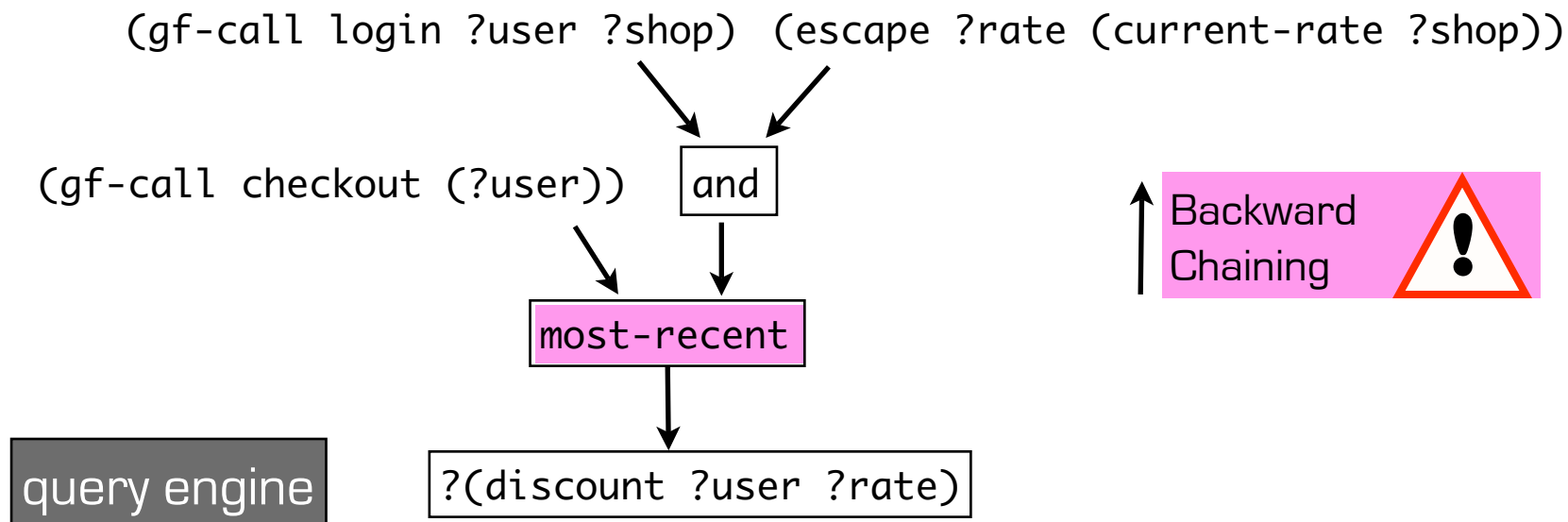


# Backward Chaining vs Rete Forward Chaining

"On every checkout, give a discount based on the rate at login time"

```
1. (set-rate *shop* 0.05)
2. (login *kris* *shop*)
3. (buy *kris* *cd*)
4. (set-rate *shop* 0.00)
5. (checkout *kris*)
```

```
(1 gf-call 'set-rate <shop> 0.05)
(2 gf-call 'login <kris> <shop>)
(3 gf-call 'buy <kris> <cd>)
(4 gf-call 'set-rate <shop> 0.00)
(5 gf-call 'checkout <kris>)
```



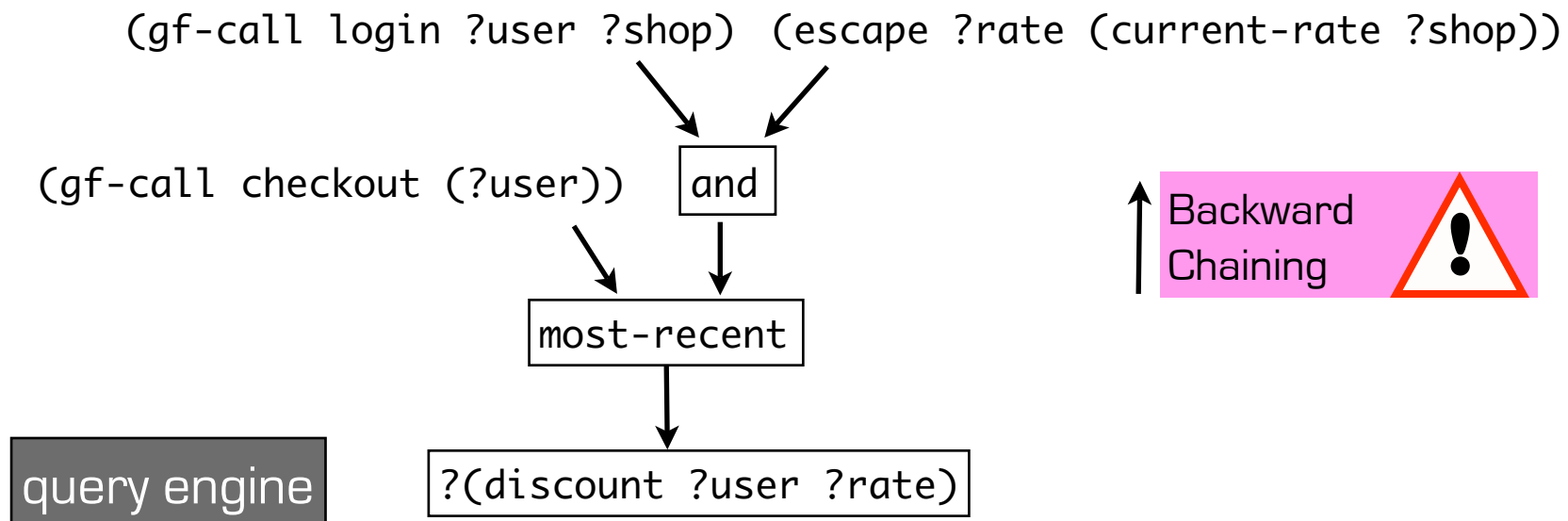


# Backward Chaining vs Rete Forward Chaining

"On every checkout, give a discount based on the rate at login time"

```
1. (set-rate *shop* 0.05)
2. (login *kris* *shop*)
3. (buy *kris* *cd*)
4. (set-rate *shop* 0.00)
5. (checkout *kris*)
```

```
(1 gf-call 'set-rate <shop> 0.05)
(2 gf-call 'login <kris> <shop>)
(3 gf-call 'buy <kris> <cd>)
(4 gf-call 'set-rate <shop> 0.00)
(5 gf-call 'checkout <kris>)
```

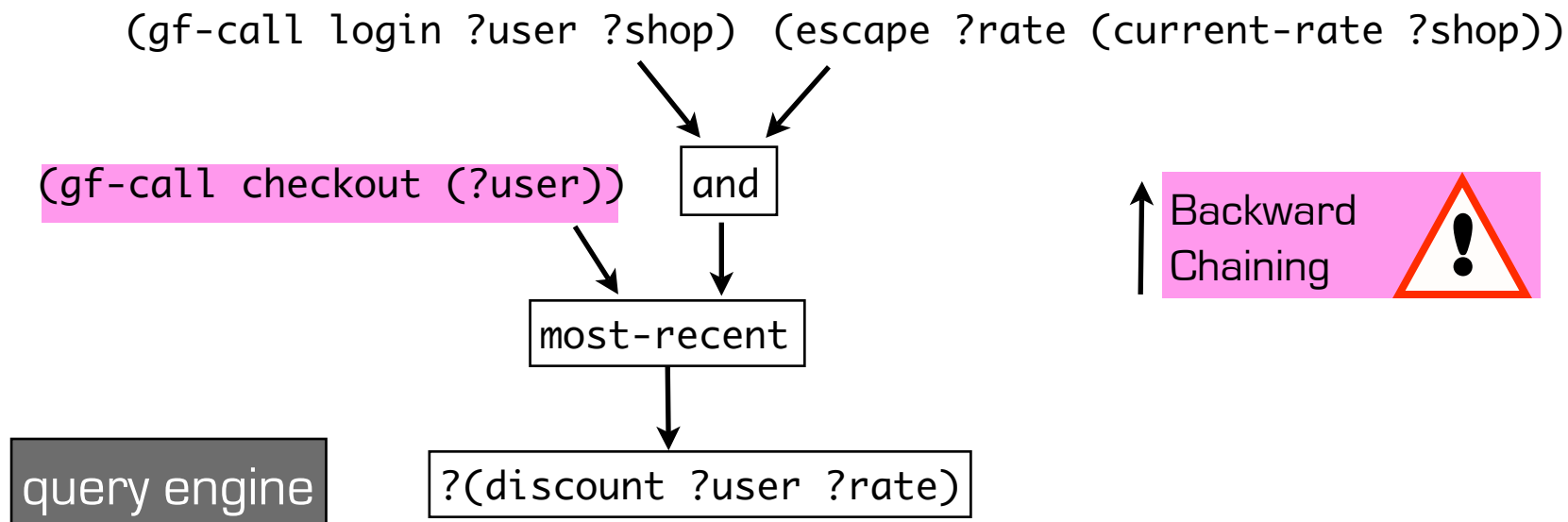


# Backward Chaining vs Rete Forward Chaining

"On every checkout, give a discount based on the rate at login time"

```
1. (set-rate *shop* 0.05)
2. (login *kris* *shop*)
3. (buy *kris* *cd*)
4. (set-rate *shop* 0.00)
5. (checkout *kris*)
```

```
(1 gf-call 'set-rate <shop> 0.05)
(2 gf-call 'login <kris> <shop>)
(3 gf-call 'buy <kris> <cd>)
(4 gf-call 'set-rate <shop> 0.00)
(5 gf-call 'checkout <kris>)
```

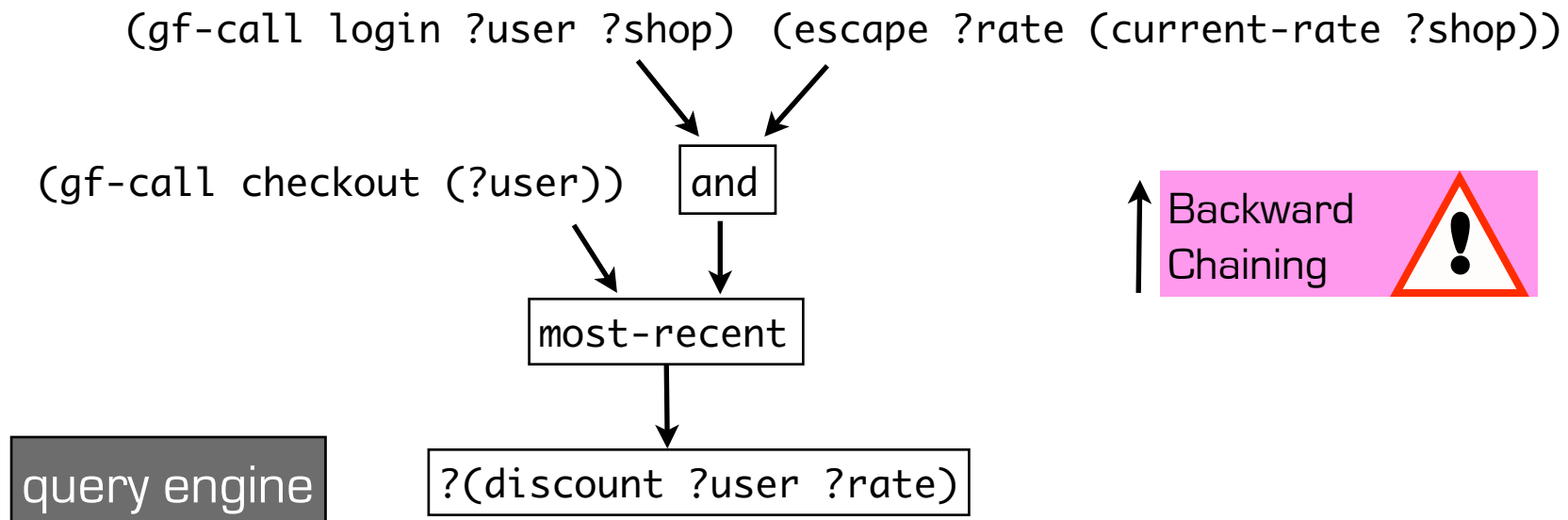


# Backward Chaining vs Rete Forward Chaining

"On every checkout, give a discount based on the rate at login time"

```
1. (set-rate *shop* 0.05)
2. (login *kris* *shop*)
3. (buy *kris* *cd*)
4. (set-rate *shop* 0.00)
5. (checkout *kris*)
```

```
(1 gf-call 'set-rate <shop> 0.05)
(2 gf-call 'login <kris> <shop>)
(3 gf-call 'buy <kris> <cd>)
(4 gf-call 'set-rate <shop> 0.00)
(5 gf-call 'checkout <kris>)
```



# Backward Chaining vs Rete Forward Chaining

"On every checkout, give a discount based on the rate at login time"

```
1. (set-rate *shop* 0.05)
2. (login *kris* *shop*)
3. (buy *kris* *cd*)
4. (set-rate *shop* 0.00)
5. (checkout *kris*)
```

```
(1 gf-call 'set-rate <shop> 0.05)
(2 gf-call 'login <kris> <shop>)
(3 gf-call 'buy <kris> <cd>)
(4 gf-call 'set-rate <shop> 0.00)
(5 gf-call 'checkout <kris>)
```

(gf-call login ?user ?shop) (escape ?rate (current-rate ?shop))

(5 gf-call 'checkout <kris>)

and

most-recent

query engine

?(discount ?user ?rate)



Backward  
Chaining



# Backward Chaining vs Rete Forward Chaining

"On every checkout, give a discount based on the rate at login time"

```
1. (set-rate *shop* 0.05)
2. (login *kris* *shop*)
3. (buy *kris* *cd*)
4. (set-rate *shop* 0.00)
5. (checkout *kris*)
```

```
(1 gf-call 'set-rate <shop> 0.05)
(2 gf-call 'login <kris> <shop>)
(3 gf-call 'buy <kris> <cd>)
(4 gf-call 'set-rate <shop> 0.00)
(5 gf-call 'checkout <kris>)
```

(gf-call login ?user ?shop) (escape ?rate (current-rate ?shop))


(5 gf-call 'checkout <kris>)

and

most-recent

query engine

?(discount ?user ?rate)

↑ Backward Chaining 

# Backward Chaining vs Rete Forward Chaining

"On every checkout, give a discount based on the rate at login time"

```
1. (set-rate *shop* 0.05)
2. (login *kris* *shop*)
3. (buy *kris* *cd*)
4. (set-rate *shop* 0.00)
5. (checkout *kris*)
```

```
(1 gf-call 'set-rate <shop> 0.05)
(2 gf-call 'login <kris> <shop>)
(3 gf-call 'buy <kris> <cd>)
(4 gf-call 'set-rate <shop> 0.00)
(5 gf-call 'checkout <kris>)
```

(gf-call login ?user ?shop) (escape ?rate (current-rate ?shop))

(5 gf-call 'checkout <kris>)

and

most-recent

query engine

?(discount ?user ?rate)



Backward  
Chaining

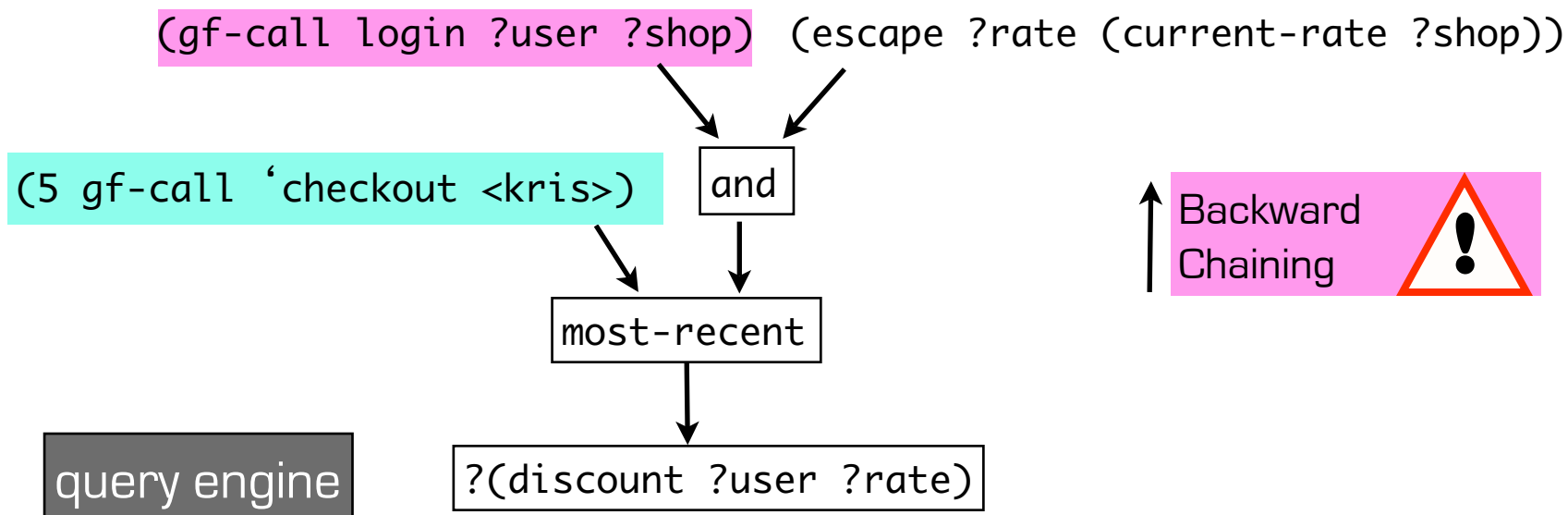


# Backward Chaining vs Rete Forward Chaining

"On every checkout, give a discount based on the rate at login time"

```
1. (set-rate *shop* 0.05)
2. (login *kris* *shop*)
3. (buy *kris* *cd*)
4. (set-rate *shop* 0.00)
5. (checkout *kris*)
```

```
(1 gf-call 'set-rate <shop> 0.05)
(2 gf-call 'login <kris> <shop>)
(3 gf-call 'buy <kris> <cd>)
(4 gf-call 'set-rate <shop> 0.00)
(5 gf-call 'checkout <kris>)
```



# Backward Chaining vs Rete Forward Chaining

"On every checkout, give a discount based on the rate at login time"

```
1. (set-rate *shop* 0.05)
2. (login *kris* *shop*)
3. (buy *kris* *cd*)
4. (set-rate *shop* 0.00)
5. (checkout *kris*)
```

```
(1 gf-call 'set-rate <shop> 0.05)
(2 gf-call 'login <kris> <shop>)
(3 gf-call 'buy <kris> <cd>)
(4 gf-call 'set-rate <shop> 0.00)
(5 gf-call 'checkout <kris>)
```

(gf-call login ?user ?shop) (escape ?rate (current-rate ?shop))

(5 gf-call 'checkout <kris>)

and

most-recent

query engine

?(discount ?user ?rate)



Backward  
Chaining





# Backward Chaining vs Rete Forward Chaining

"On every checkout, give a discount based on the rate at login time"

```
1. (set-rate *shop* 0.05)
2. (login *kris* *shop*)
3. (buy *kris* *cd*)
4. (set-rate *shop* 0.00)
5. (checkout *kris*)
```

```
(1 gf-call 'set-rate <shop> 0.05)
(2 gf-call 'login <kris> <shop>)
(3 gf-call 'buy <kris> <cd>)
(4 gf-call 'set-rate <shop> 0.00)
(5 gf-call 'checkout <kris>)
```

(2 gf-call 'login <kris> <shop>) (escape ?rate (current-rate ?shop))

(5 gf-call 'checkout <kris>)

and

most-recent

query engine

?(discount ?user ?rate)



Backward  
Chaining



# Backward Chaining vs Rete Forward Chaining

"On every checkout, give a discount based on the rate at login time"

```
1. (set-rate *shop* 0.05)
2. (login *kris* *shop*)
3. (buy *kris* *cd*)
4. (set-rate *shop* 0.00)
5. (checkout *kris*)
```

```
(1 gf-call 'set-rate <shop> 0.05)
(2 gf-call 'login <kris> <shop>)
(3 gf-call 'buy <kris> <cd>)
(4 gf-call 'set-rate <shop> 0.00)
(5 gf-call 'checkout <kris>)
```

```
(2 gf-call 'login <kris> <shop>)
```

```
(escape ?rate (current-rate ?shop))
```

```
(5 gf-call 'checkout <kris>)
```

and

most-recent

query engine

```
?(discount ?user ?rate)
```



Backward  
Chaining



# Backward Chaining vs Rete Forward Chaining

"On every checkout, give a discount based on the rate at login time"

```
1. (set-rate *shop* 0.05)
2. (login *kris* *shop*)
3. (buy *kris* *cd*)
4. (set-rate *shop* 0.00)
5. (checkout *kris*)
```

```
(1 gf-call 'set-rate <shop> 0.05)
(2 gf-call 'login <kris> <shop>)
(3 gf-call 'buy <kris> <cd>)
(4 gf-call 'set-rate <shop> 0.00)
(5 gf-call 'checkout <kris>)
```

(2 gf-call 'login <kris> <shop>) (escape ?rate (current-rate ?shop))

(5 gf-call 'checkout <kris>)

and

most-recent

query engine

?(discount ?user ?rate)



Backward  
Chaining



# Backward Chaining vs Rete Forward Chaining

"On every checkout, give a discount based on the rate at login time"

```
1. (set-rate *shop* 0.05)
2. (login *kris* *shop*)
3. (buy *kris* *cd*)
4. (set-rate *shop* 0.00)
5. (checkout *kris*)
```

```
(1 gf-call 'set-rate <shop> 0.05)
(2 gf-call 'login <kris> <shop>)
(3 gf-call 'buy <kris> <cd>)
(4 gf-call 'set-rate <shop> 0.00)
(5 gf-call 'checkout <kris>)
```

```
(2 gf-call 'login <kris> <shop>)
```

```
(escape 0.00 (current-rate <shop>))
```

```
(5 gf-call 'checkout <kris>)
```

and

most-recent

query engine

?(discount ?user ?rate)



Backward  
Chaining



# Backward Chaining vs Rete Forward Chaining

"On every checkout, give a discount based on the rate at login time"

```
1. (set-rate *shop* 0.05)
2. (login *kris* *shop*)
3. (buy *kris* *cd*)
4. (set-rate *shop* 0.00)
5. (checkout *kris*)
```

```
(1 gf-call 'set-rate <shop> 0.05)
(2 gf-call 'login <kris> <shop>)
(3 gf-call 'buy <kris> <cd>)
(4 gf-call 'set-rate <shop> 0.00)
(5 gf-call 'checkout <kris>)
```

```
(2 gf-call 'login <kris> <shop>)
```

```
(escape 0.00 (current-rate <shop>))
```

```
(5 gf-call 'checkout <kris>)
```

and

most-recent

query engine

?(discount ?user ?rate)

↑ Backward Chaining



# Backward Chaining vs Rete Forward Chaining

"On every checkout, give a discount based on the rate at login time"

```
1. (set-rate *shop* 0.05)
2. (login *kris* *shop*)
3. (buy *kris* *cd*)
4. (set-rate *shop* 0.00)
5. (checkout *kris*)
```

```
(1 gf-call 'set-rate <shop> 0.05)
(2 gf-call 'login <kris> <shop>)
(3 gf-call 'buy <kris> <cd>)
(4 gf-call 'set-rate <shop> 0.00)
(5 gf-call 'checkout <kris>)
```

```
(2 gf-call 'login <kris> <shop>)
```

```
(escape 0.00 (current-rate <shop>))
```


```
(5 gf-call 'checkout <kris>)
```

and

most-recent

query engine

?(discount ?user ?rate)

↑ Backward Chaining 

# Backward Chaining vs Rete Forward Chaining

"On every checkout, give a discount based on the rate at login time"

```
1. (set-rate *shop* 0.05)
2. (login *kris* *shop*)
3. (buy *kris* *cd*)
4. (set-rate *shop* 0.00)
5. (checkout *kris*)
```

```
(1 gf-call 'set-rate <shop> 0.05)
(2 gf-call 'login <kris> <shop>)
(3 gf-call 'buy <kris> <cd>)
(4 gf-call 'set-rate <shop> 0.00)
(5 gf-call 'checkout <kris>)
```

```
(2 gf-call 'login <kris> <shop>)
```

```
(escape 0.00 (current-rate <shop>))
```

```
(5 gf-call 'checkout <kris>)
```

and

most-recent

query engine

```
(discount <kris> 0.00)
```



Backward  
Chaining



# Backward Chaining vs Rete Forward Chaining

"On every checkout, give a discount based on the rate at login time"

```
1. (set-rate *shop* 0.05)
2. (login *kris* *shop*)
3. (buy *kris* *cd*)
4. (set-rate *shop* 0.00)
5. (checkout *kris*)
```

```
(1 gf-call 'set-rate <shop> 0.05)
(2 gf-call 'login <kris> <shop>)
(3 gf-call 'buy <kris> <cd>)
(4 gf-call 'set-rate <shop> 0.00)
(5 gf-call 'checkout <kris>)
```

```
(2 gf-call 'login <kris> <shop>) (escape 0.00 (current-rate <shop>))
```



```
(5 gf-call 'checkout <kris>)
```

and

most-recent

query engine

```
(discount <kris> 0.00)
```



Backward  
Chaining

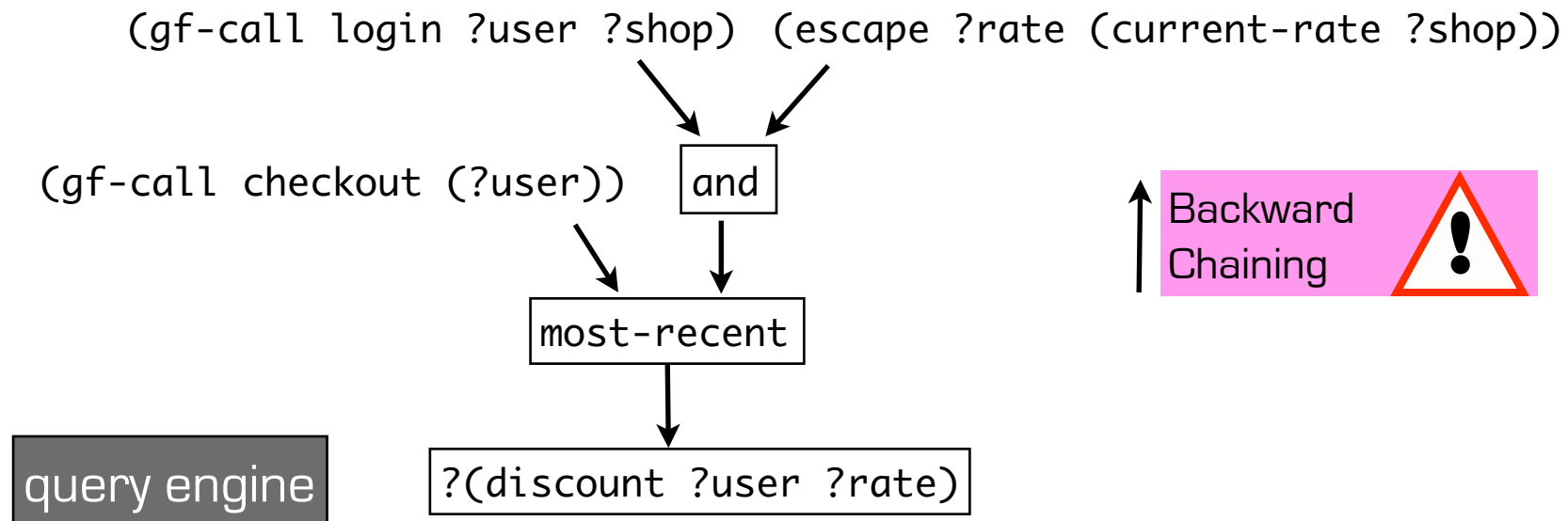




# Backward Chaining vs Rete Forward Chaining

"On every checkout, give a discount based on the rate at login time"

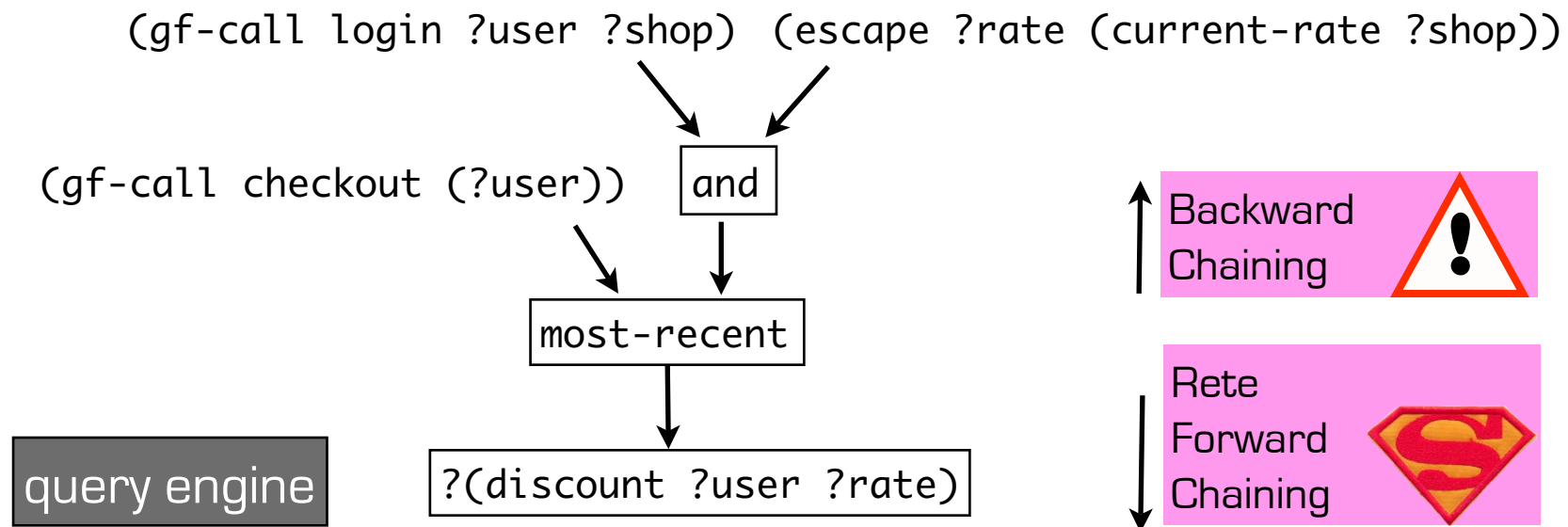
```
(set-rate *shop* 0.05)
(login *kris* *shop*)
(buy *kris* *cd*)
(set-rate *shop* 0.00)
(checkout *kris*)
```



# Backward Chaining vs Rete Forward Chaining

"On every checkout, give a discount  
based on the rate at login time"

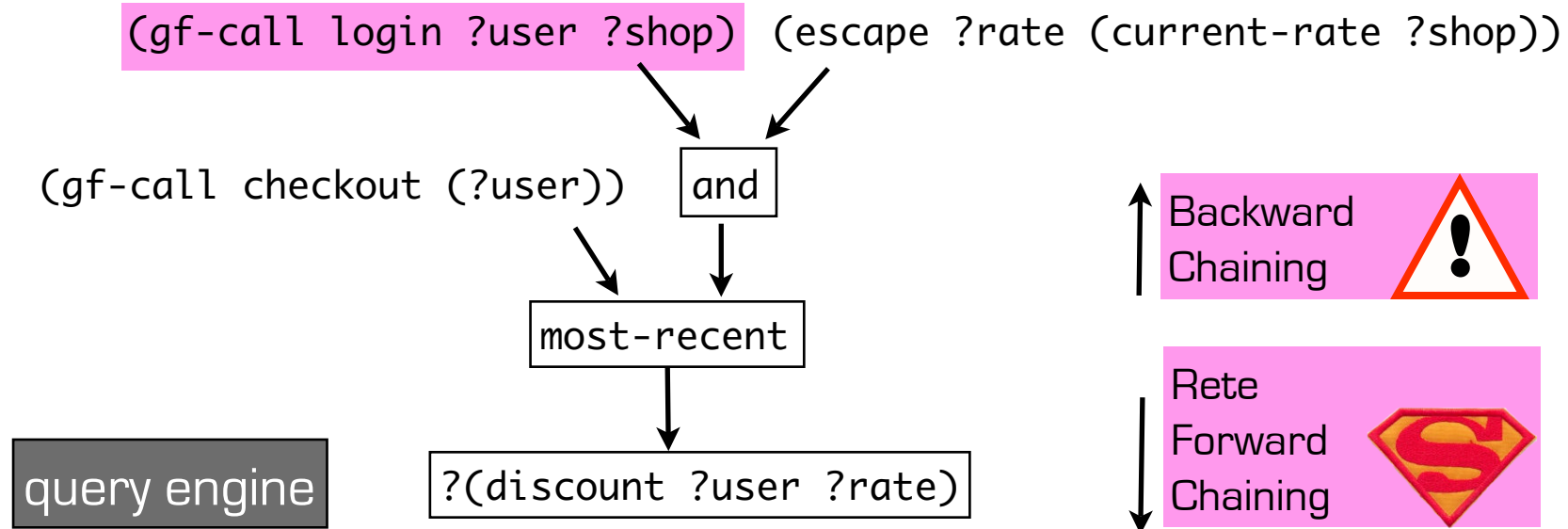
```
(set-rate *shop* 0.05)
(login *kris* *shop*)
(buy *kris* *cd*)
(set-rate *shop* 0.00)
(checkout *kris*)
```



# Backward Chaining vs Rete Forward Chaining

"On every checkout, give a discount  
based on the rate at login time"

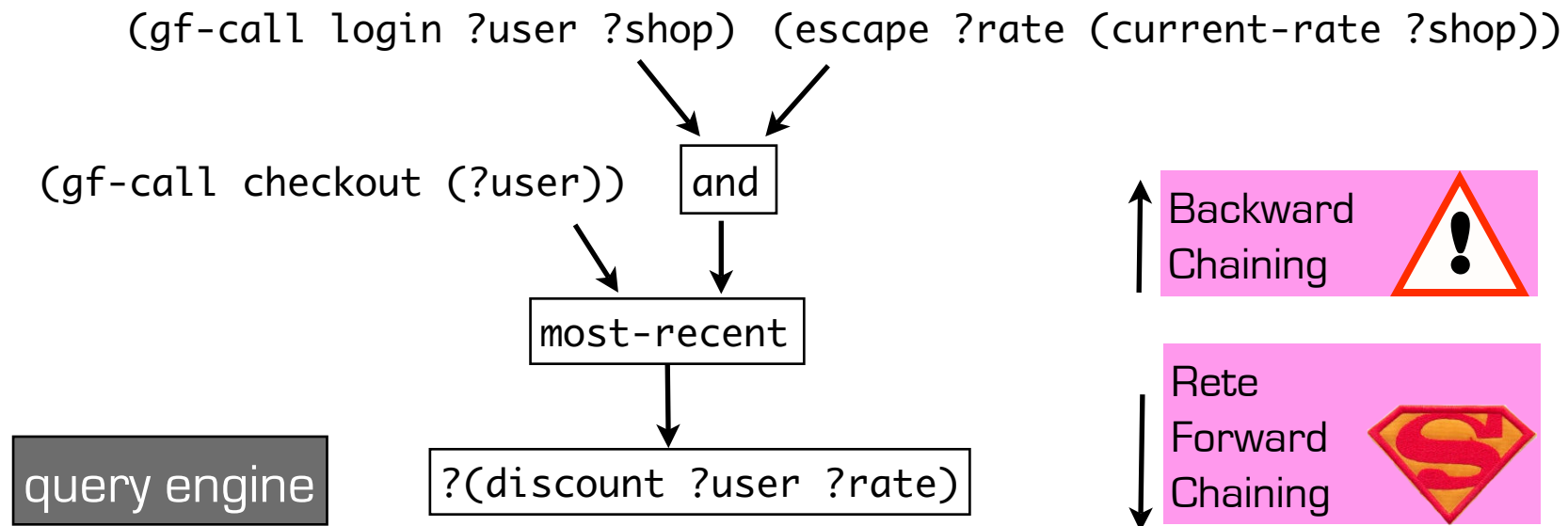
```
(set-rate *shop* 0.05)
(login *kris* *shop*)
(buy *kris* *cd*)
(set-rate *shop* 0.00)
(checkout *kris*)
```



# Backward Chaining vs Rete Forward Chaining

"On every checkout, give a discount  
based on the rate at login time"

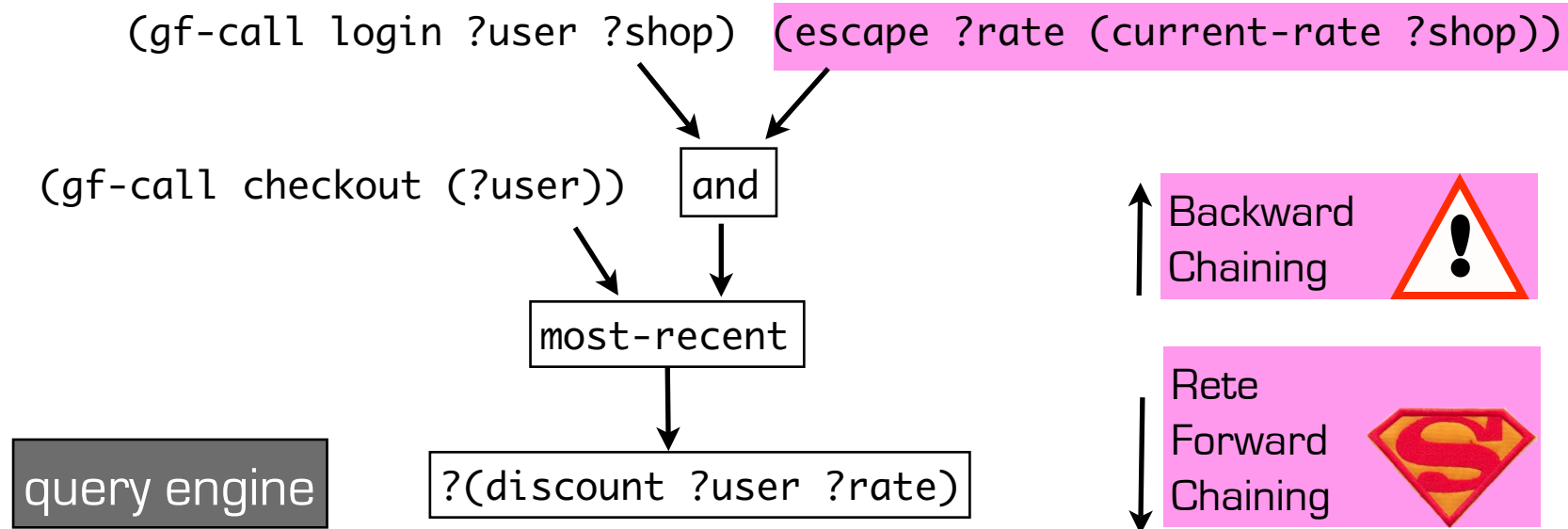
```
(set-rate *shop* 0.05)
(login *kris* *shop*)
(buy *kris* *cd*)
(set-rate *shop* 0.00)
(checkout *kris*)
```



# Backward Chaining vs Rete Forward Chaining

"On every checkout, give a discount based on the rate at login time"

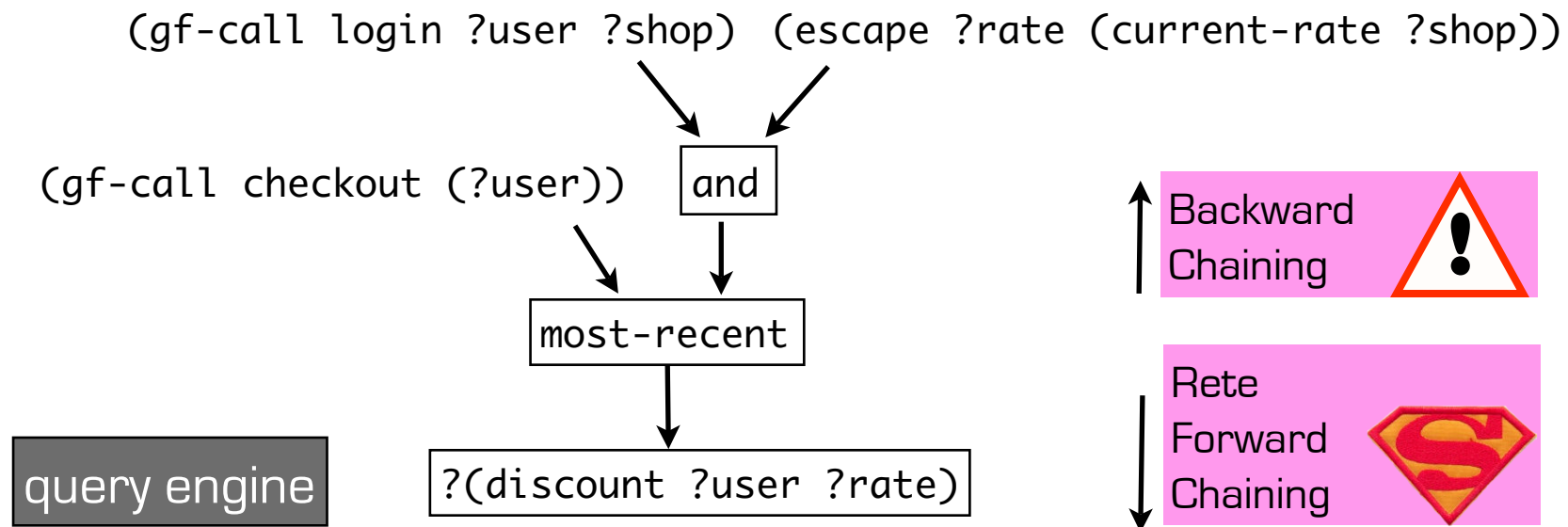
```
(set-rate *shop* 0.05)
(login *kris* *shop*)
(buy *kris* *cd*)
(set-rate *shop* 0.00)
(checkout *kris*)
```



# Backward Chaining vs Rete Forward Chaining

"On every checkout, give a discount  
based on the rate at login time"

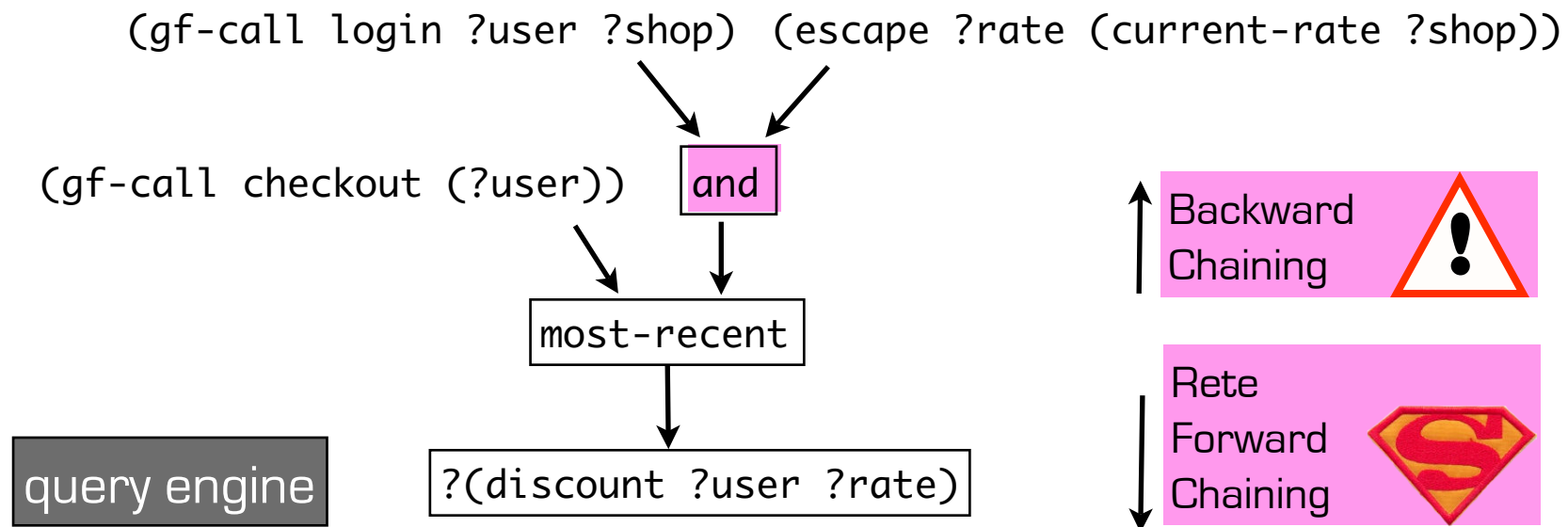
```
(set-rate *shop* 0.05)
(login *kris* *shop*)
(buy *kris* *cd*)
(set-rate *shop* 0.00)
(checkout *kris*)
```



# Backward Chaining vs Rete Forward Chaining

"On every checkout, give a discount based on the rate at login time"

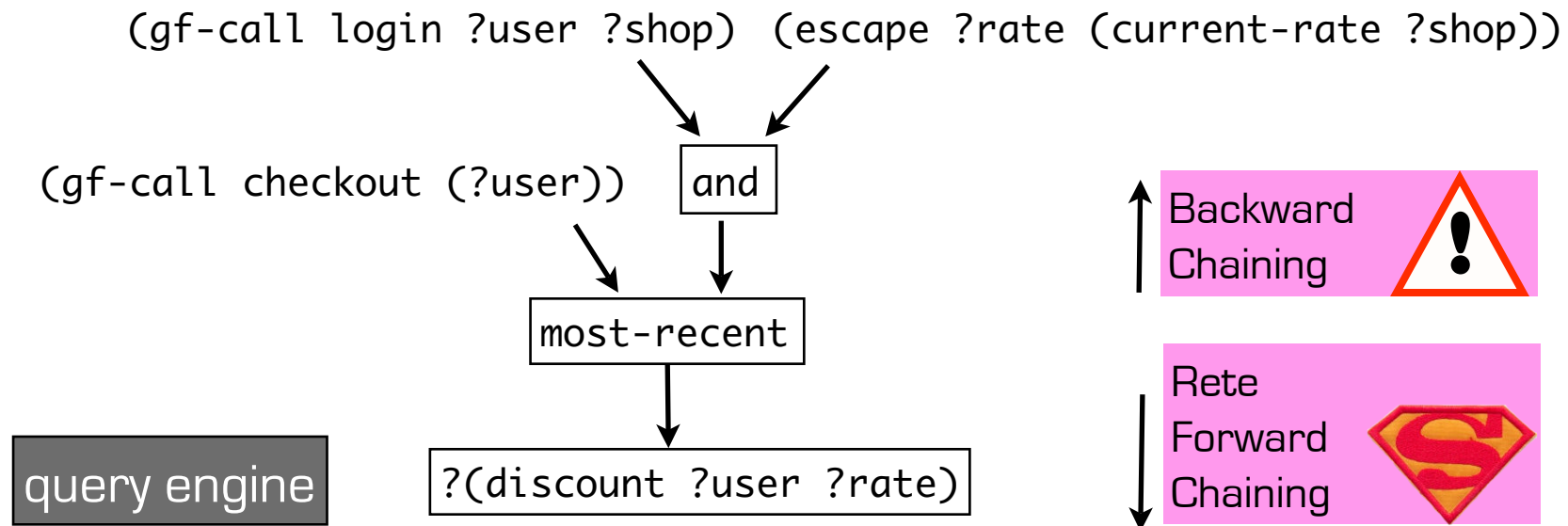
```
(set-rate *shop* 0.05)
(login *kris* *shop*)
(buy *kris* *cd*)
(set-rate *shop* 0.00)
(checkout *kris*)
```



# Backward Chaining vs Rete Forward Chaining

"On every checkout, give a discount  
based on the rate at login time"

```
(set-rate *shop* 0.05)
(login *kris* *shop*)
(buy *kris* *cd*)
(set-rate *shop* 0.00)
(checkout *kris*)
```

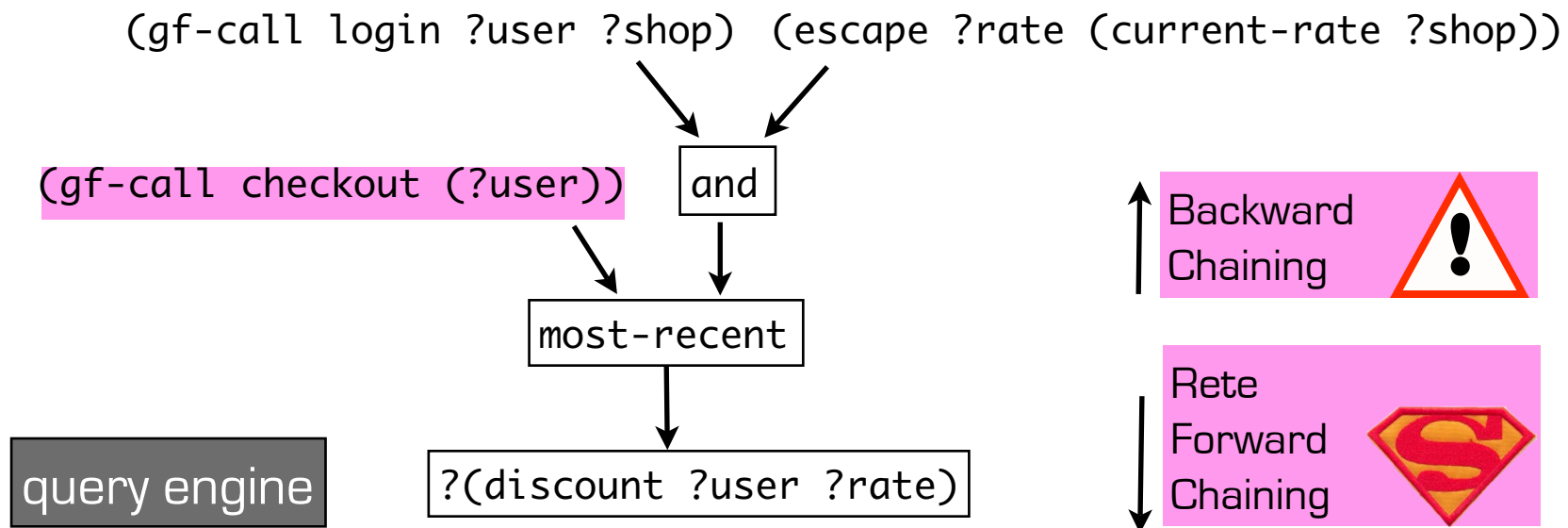




# Backward Chaining vs Rete Forward Chaining

"On every checkout, give a discount  
based on the rate at login time"

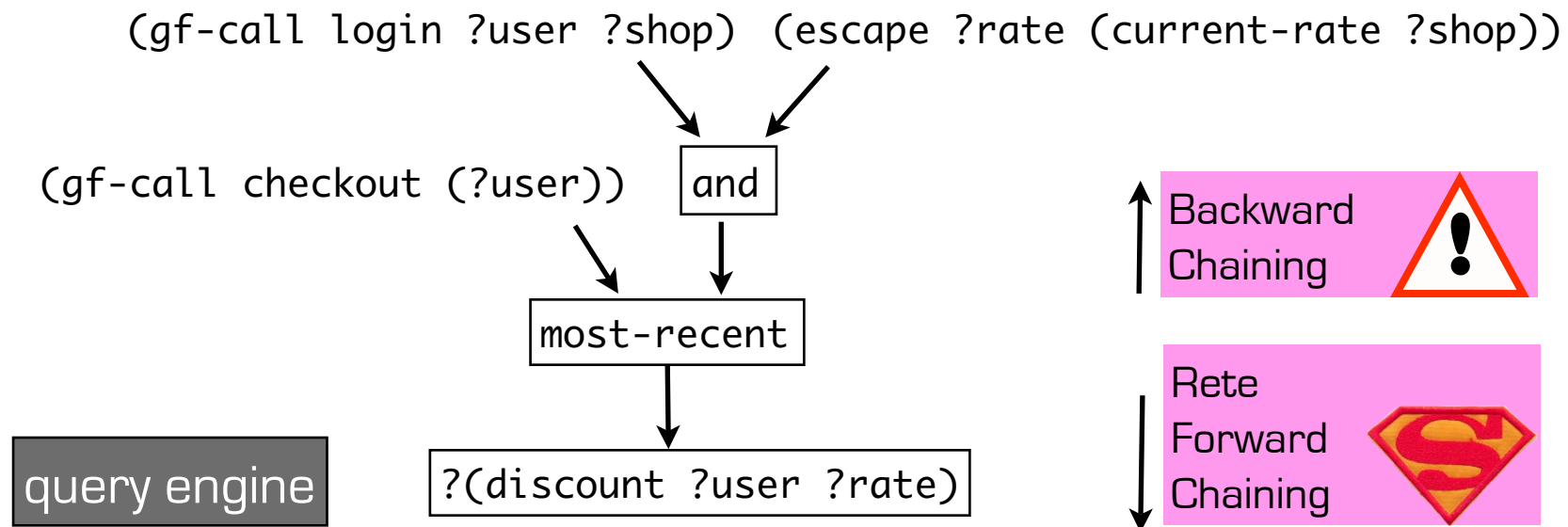
```
(set-rate *shop* 0.05)
(login *kris* *shop*)
(buy *kris* *cd*)
(set-rate *shop* 0.00)
(checkout *kris*)
```



# Backward Chaining vs Rete Forward Chaining

"On every checkout, give a discount  
based on the rate at login time"

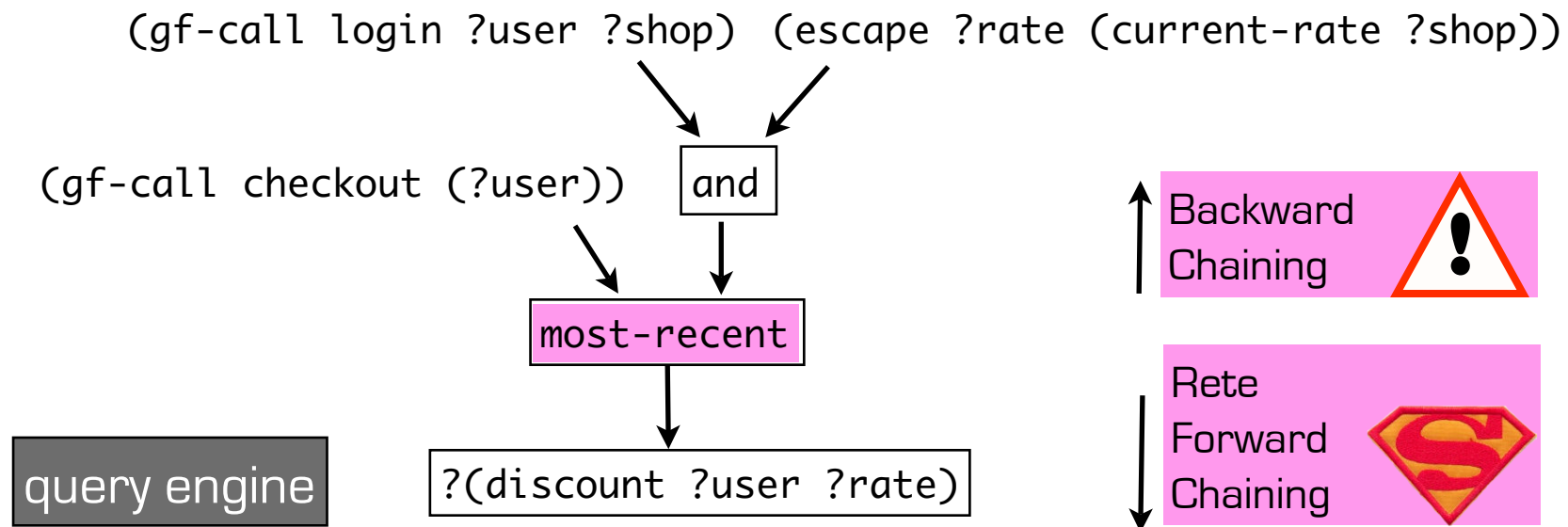
```
(set-rate *shop* 0.05)  
(login *kris* *shop*)  
(buy *kris* *cd*)  
(set-rate *shop* 0.00)  
(checkout *kris*)
```



# Backward Chaining vs Rete Forward Chaining

"On every checkout, give a discount  
based on the rate at login time"

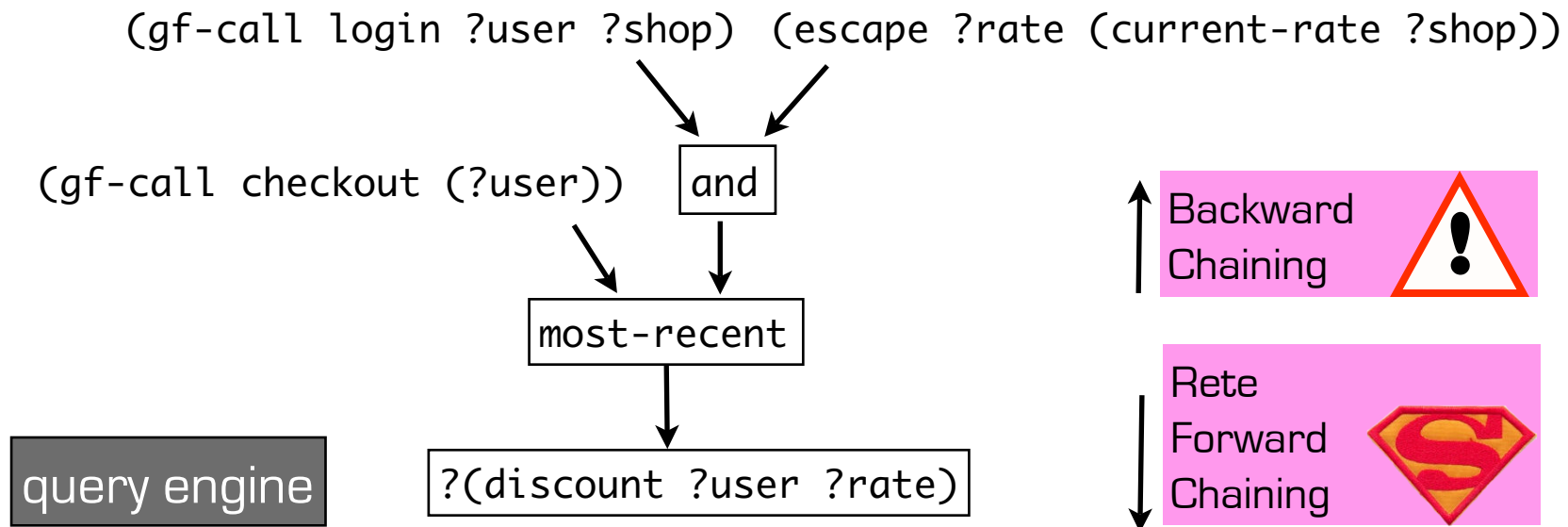
```
(set-rate *shop* 0.05)
(login *kris* *shop*)
(buy *kris* *cd*)
(set-rate *shop* 0.00)
(checkout *kris*)
```



# Backward Chaining vs Rete Forward Chaining

"On every checkout, give a discount  
based on the rate at login time"

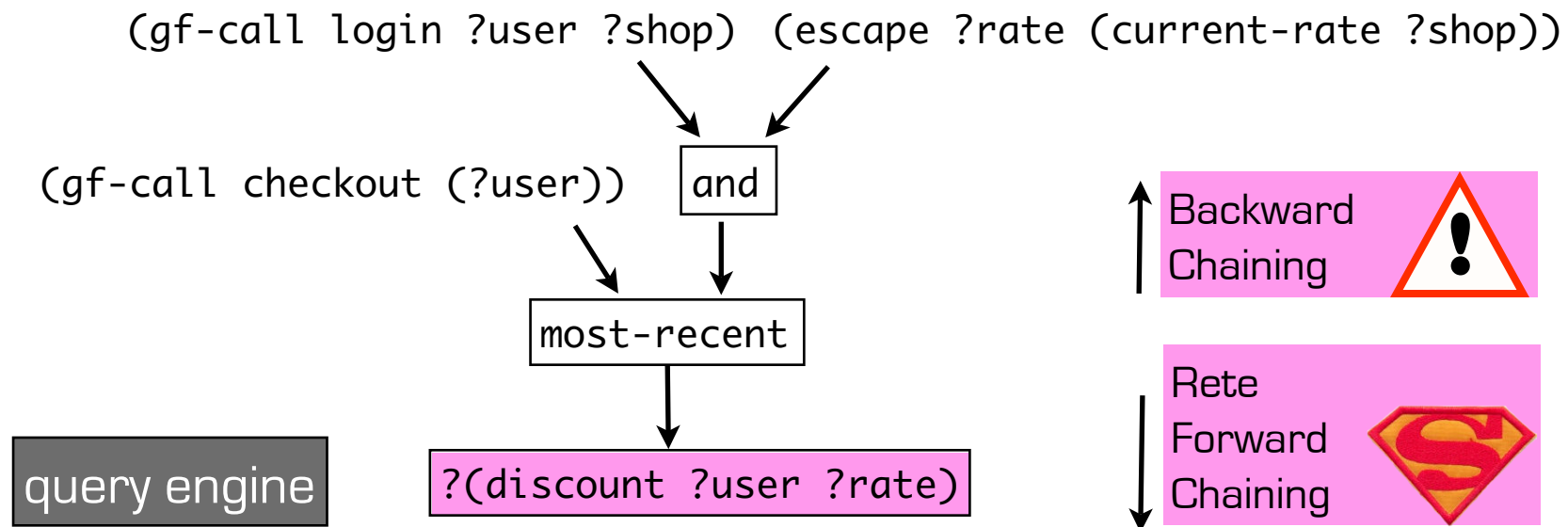
```
(set-rate *shop* 0.05)
(login *kris* *shop*)
(buy *kris* *cd*)
(set-rate *shop* 0.00)
(checkout *kris*)
```



# Backward Chaining vs Rete Forward Chaining

"On every checkout, give a discount  
based on the rate at login time"

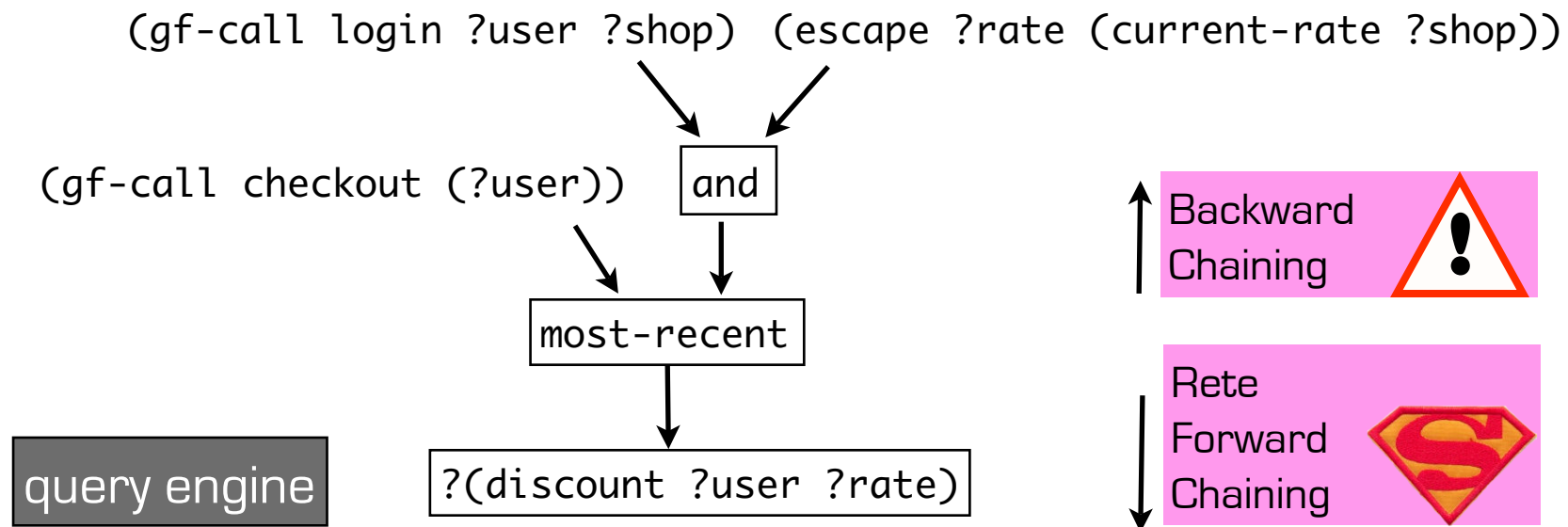
```
(set-rate *shop* 0.05)
(login *kris* *shop*)
(buy *kris* *cd*)
(set-rate *shop* 0.00)
(checkout *kris*)
```



# Backward Chaining vs Rete Forward Chaining

"On every checkout, give a discount  
based on the rate at login time"

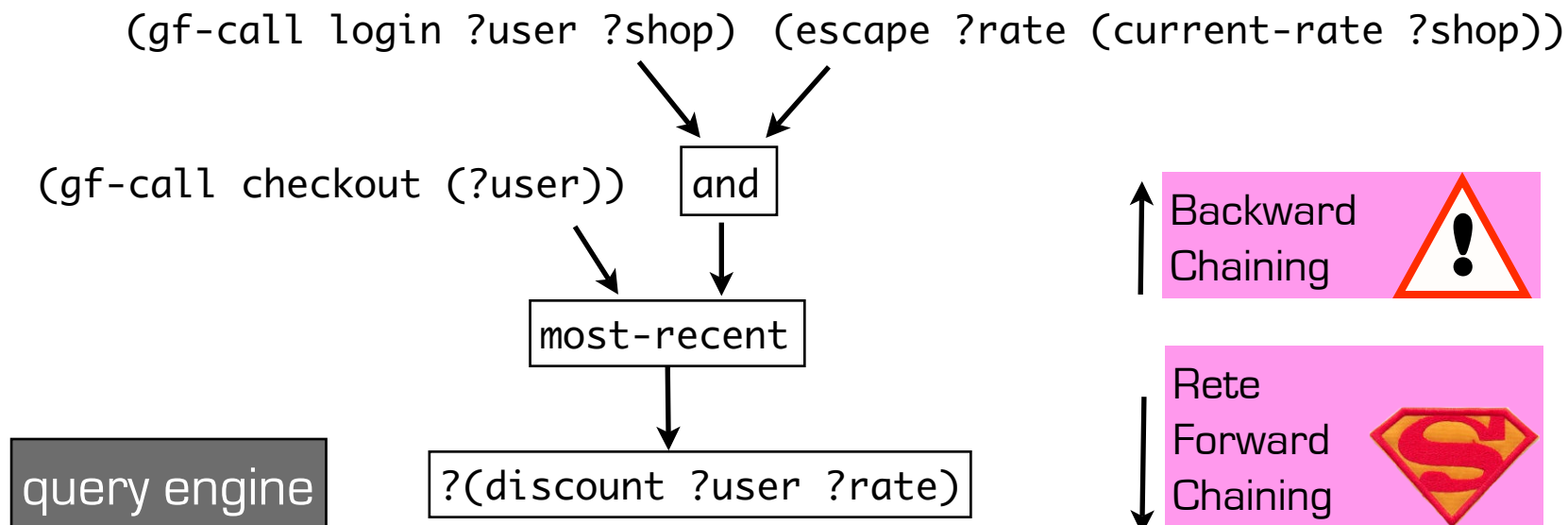
```
(set-rate *shop* 0.05)
(login *kris* *shop*)
(buy *kris* *cd*)
(set-rate *shop* 0.00)
(checkout *kris*)
```



# Backward Chaining vs Rete Forward Chaining

"On every checkout, give a discount based on the rate at login time"

```
1. (set-rate *shop* 0.05)
   (login *kris* *shop*)
   (buy *kris* *cd*)
   (set-rate *shop* 0.00)
   (checkout *kris*)
```

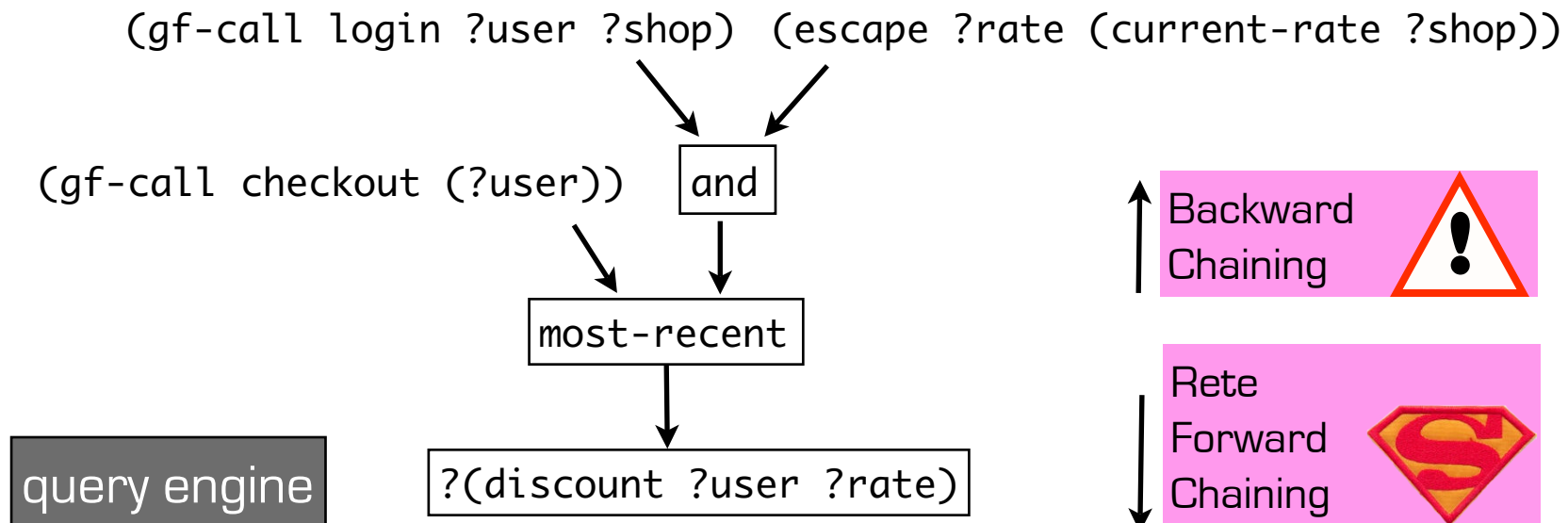


# Backward Chaining vs Rete Forward Chaining

"On every checkout, give a discount based on the rate at login time"

```
1. (set-rate *shop* 0.05)
   (login *kris* *shop*)
   (buy *kris* *cd*)
   (set-rate *shop* 0.00)
   (checkout *kris*)
```

```
(1 gf-call 'set-rate <shop> 0.05)
```





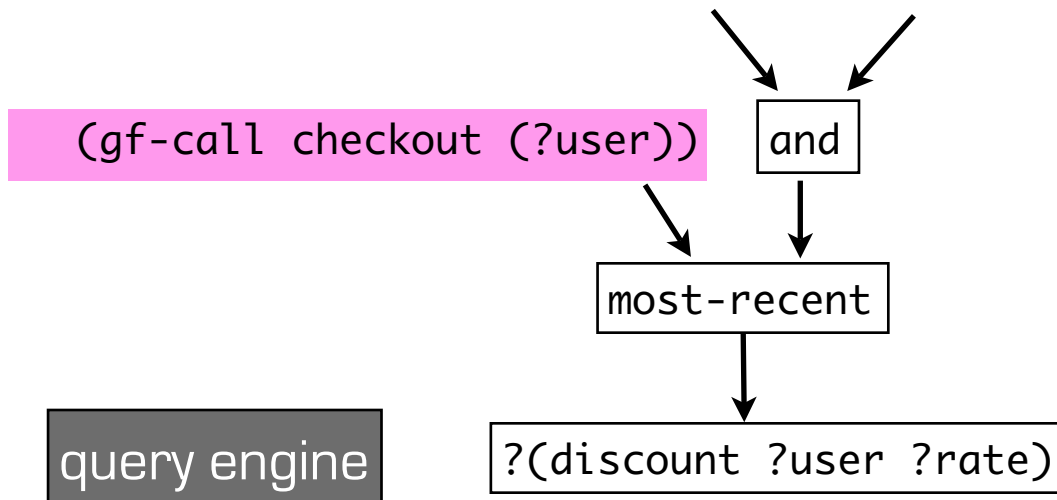
# Backward Chaining vs Rete Forward Chaining


"On every checkout, give a discount based on the rate at login time"

```
1. (set-rate *shop* 0.05)
   (login *kris* *shop*)
   (buy *kris* *cd*)
   (set-rate *shop* 0.00)
   (checkout *kris*)
```

```
(1 gf-call 'set-rate <shop> 0.05)
```

```
(gf-call login ?user ?shop) (escape ?rate (current-rate ?shop))
```



↑ Backward Chaining 

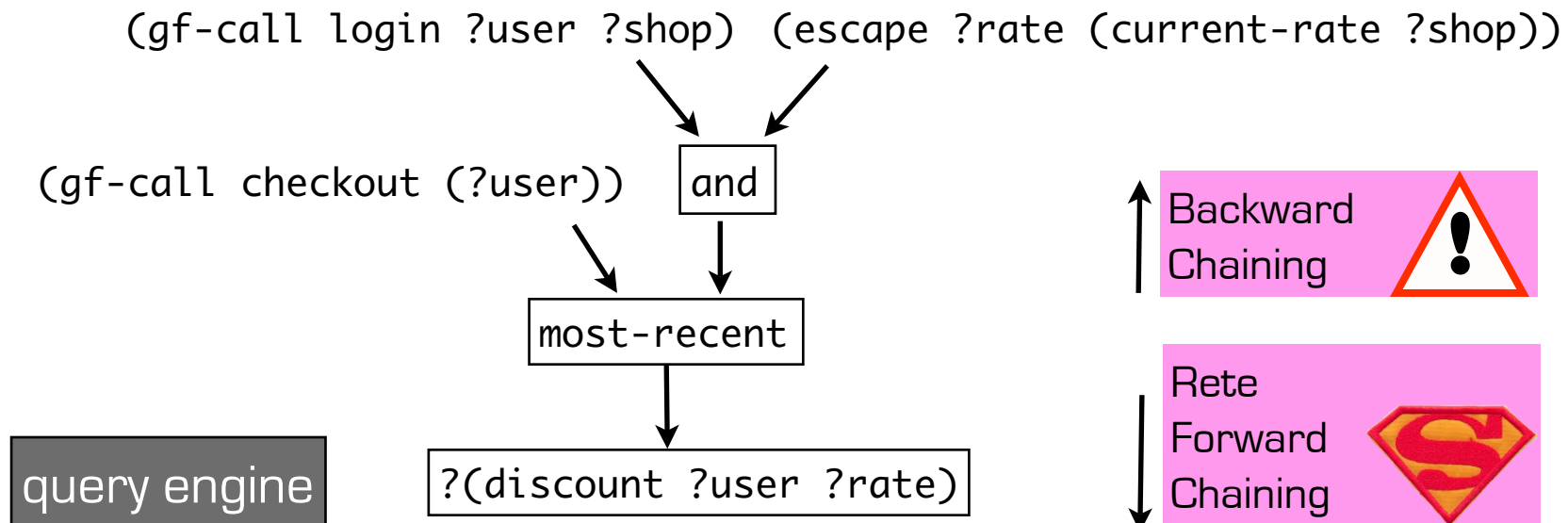
↓ Rete Forward Chaining 

# Backward Chaining vs Rete Forward Chaining

"On every checkout, give a discount based on the rate at login time"

```
1. (set-rate *shop* 0.05)
   (login *kris* *shop*)
   (buy *kris* *cd*)
   (set-rate *shop* 0.00)
   (checkout *kris*)
```

```
(1 gf-call 'set-rate <shop> 0.05)
```

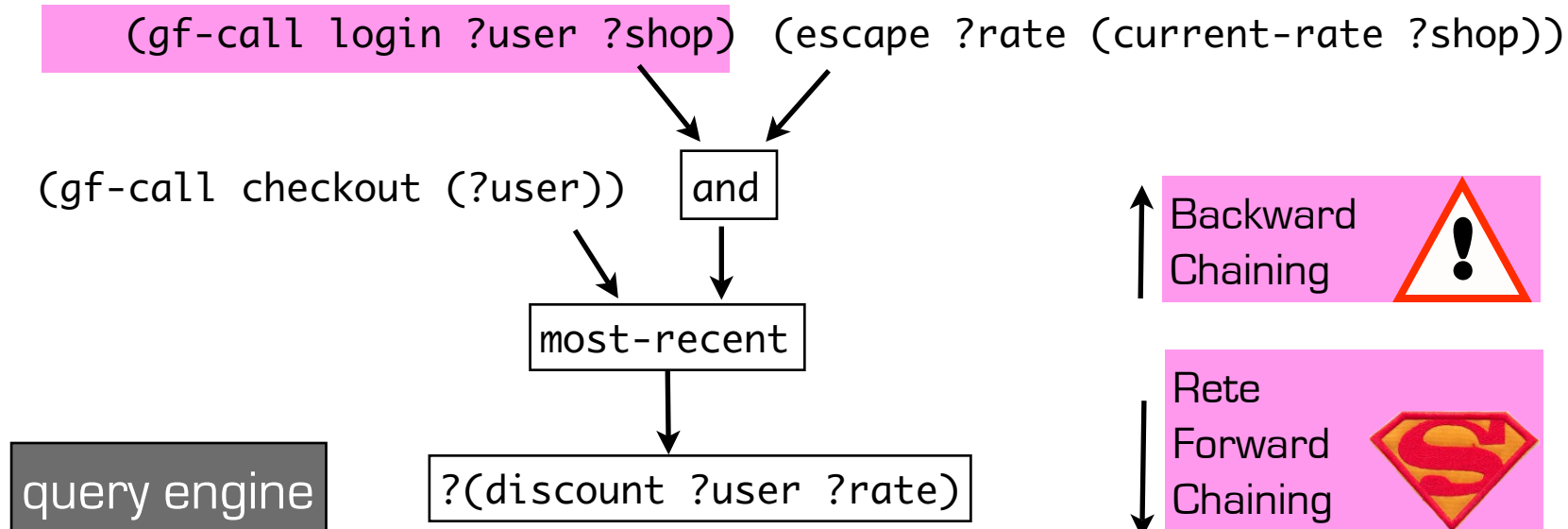


# Backward Chaining vs Rete Forward Chaining

"On every checkout, give a discount  
based on the rate at login time"

```
1. (set-rate *shop* 0.05)  
   (login *kris* *shop*)  
   (buy *kris* *cd*)  
   (set-rate *shop* 0.00)  
   (checkout *kris*)
```

```
(1 gf-call 'set-rate <shop> 0.05)
```

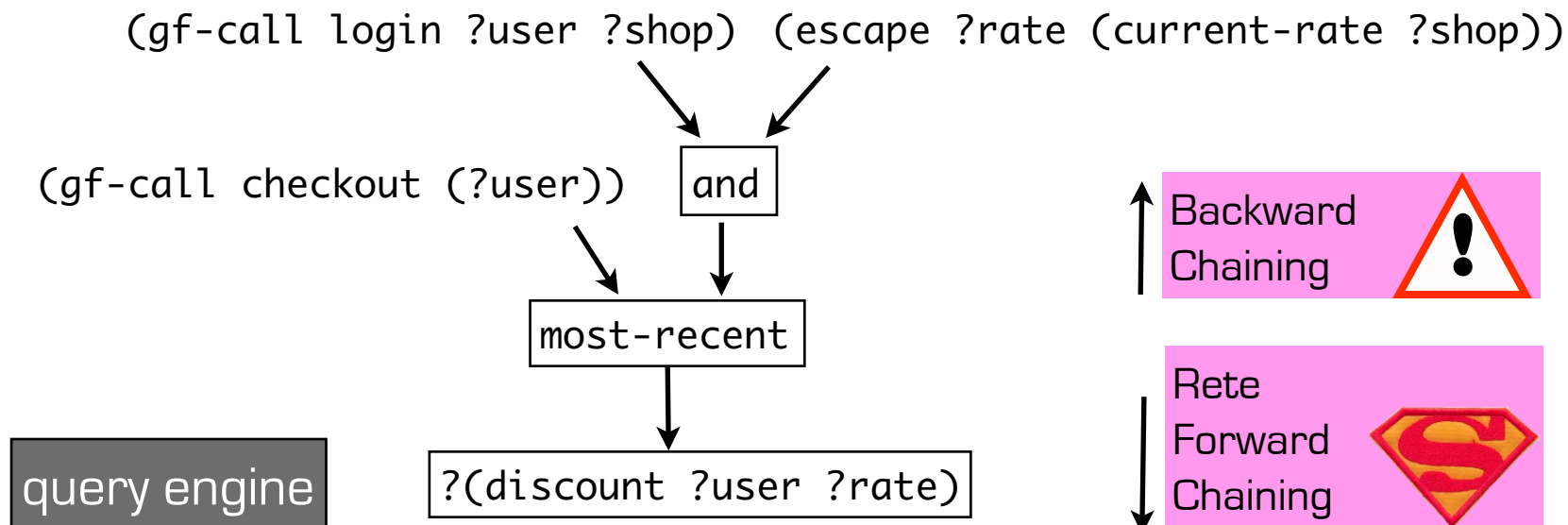


# Backward Chaining vs Rete Forward Chaining

"On every checkout, give a discount  
based on the rate at login time"

```
1. (set-rate *shop* 0.05)  
   (login *kris* *shop*)  
   (buy *kris* *cd*)  
   (set-rate *shop* 0.00)  
   (checkout *kris*)
```

```
(1 gf-call 'set-rate <shop> 0.05)
```

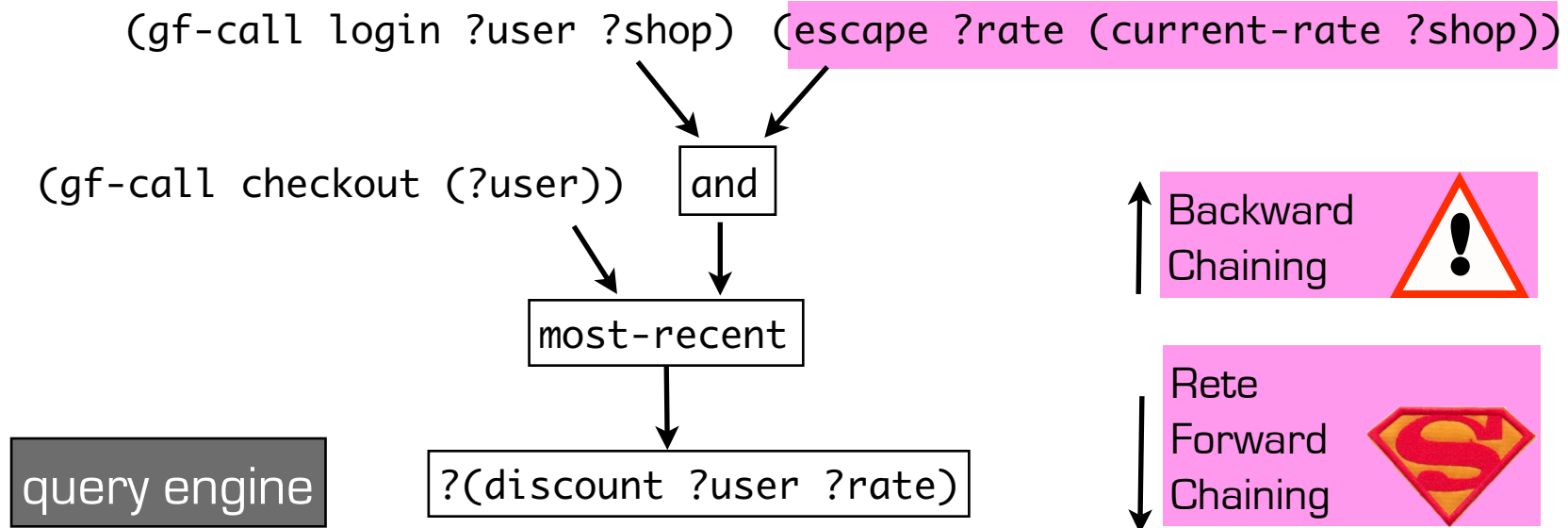


# Backward Chaining vs Rete Forward Chaining

"On every checkout, give a discount based on the rate at login time"

```
1. (set-rate *shop* 0.05)
   (login *kris* *shop*)
   (buy *kris* *cd*)
   (set-rate *shop* 0.00)
   (checkout *kris*)
```

```
(1 gf-call 'set-rate <shop> 0.05)
```

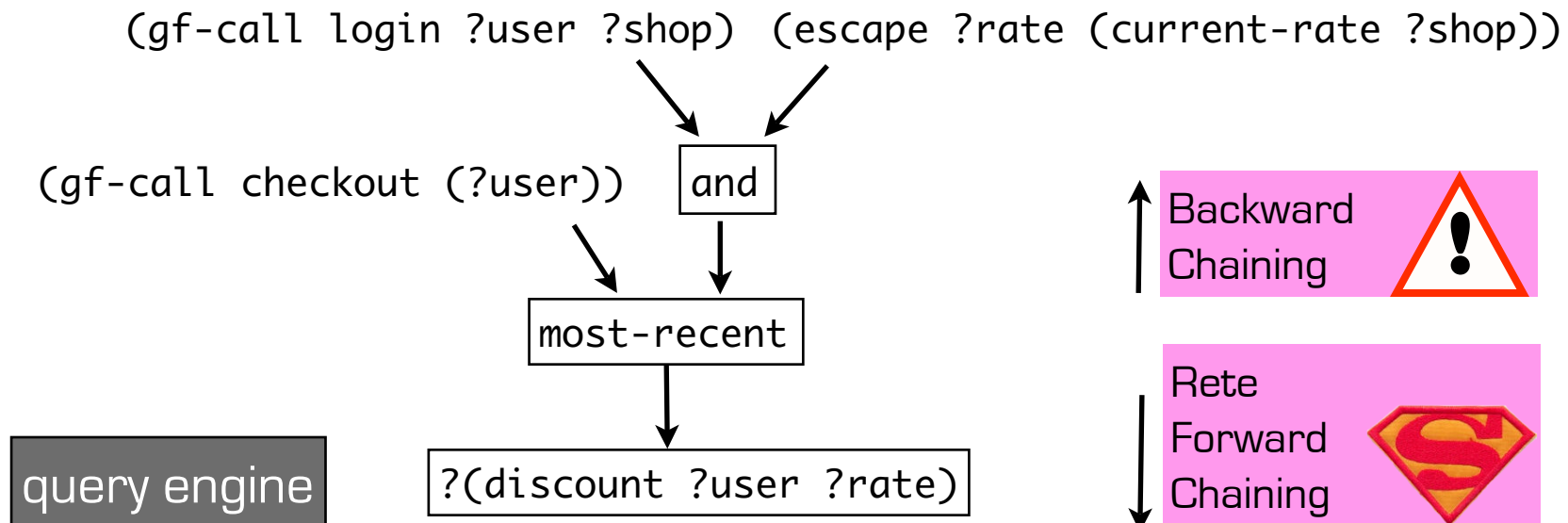


# Backward Chaining vs Rete Forward Chaining

"On every checkout, give a discount  
based on the rate at login time"

```
1. (set-rate *shop* 0.05)  
   (login *kris* *shop*)  
   (buy *kris* *cd*)  
   (set-rate *shop* 0.00)  
   (checkout *kris*)
```

```
(1 gf-call 'set-rate <shop> 0.05)
```

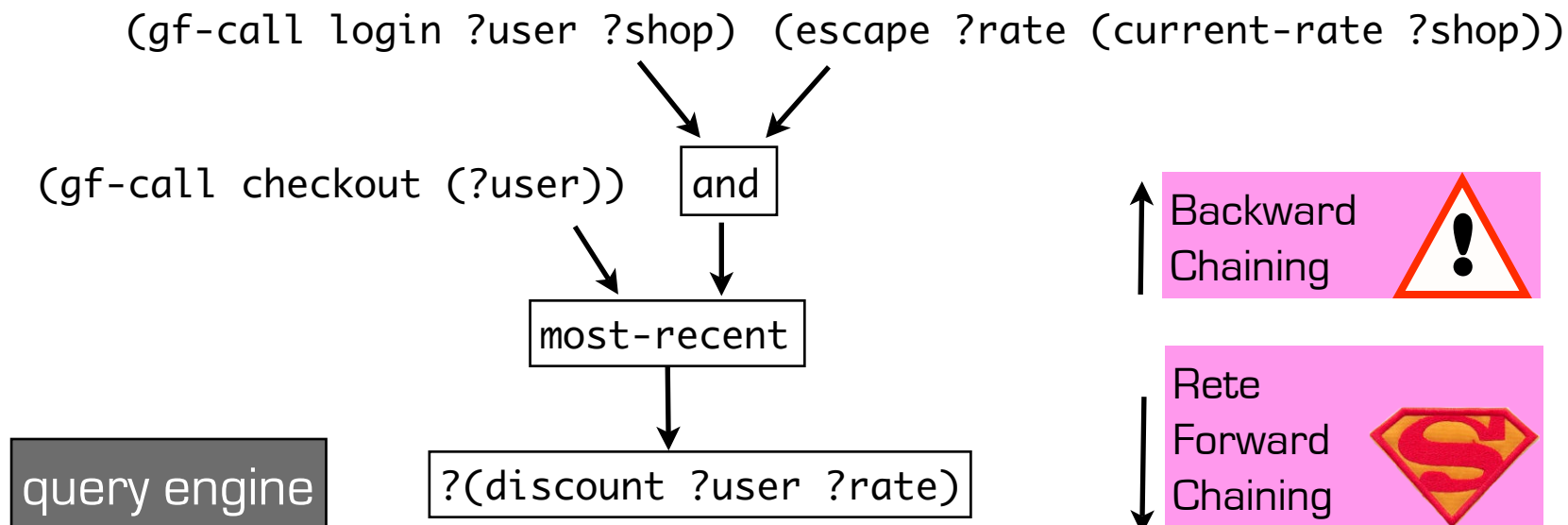


# Backward Chaining vs Rete Forward Chaining

"On every checkout, give a discount  
based on the rate at login time"

```
1. (set-rate *shop* 0.05)
2. (login *kris* *shop*)
   (buy *kris* *cd*)
   (set-rate *shop* 0.00)
   (checkout *kris*)
```

```
(1 gf-call 'set-rate <shop> 0.05)
```

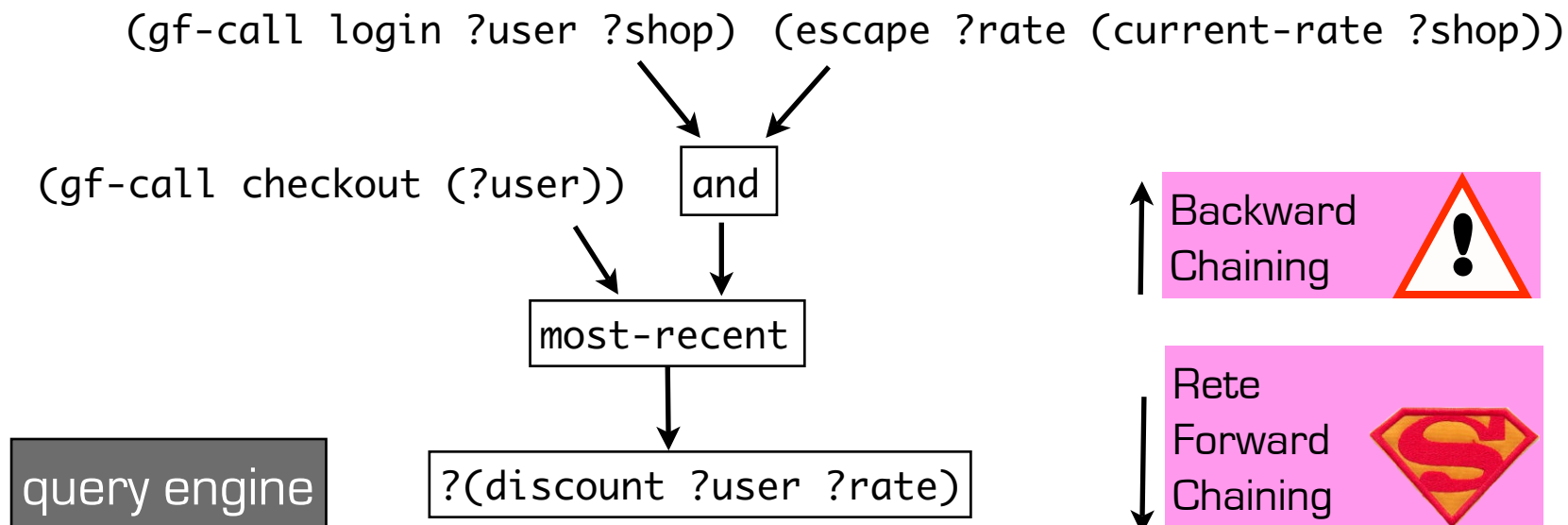


# Backward Chaining vs Rete Forward Chaining

"On every checkout, give a discount  
based on the rate at login time"

```
1. (set-rate *shop* 0.05)
2. (login *kris* *shop*)
   (buy *kris* *cd*)
   (set-rate *shop* 0.00)
   (checkout *kris*)
```

```
(1 gf-call 'set-rate <shop> 0.05)
(2 gf-call 'login <kris> <shop>)
```





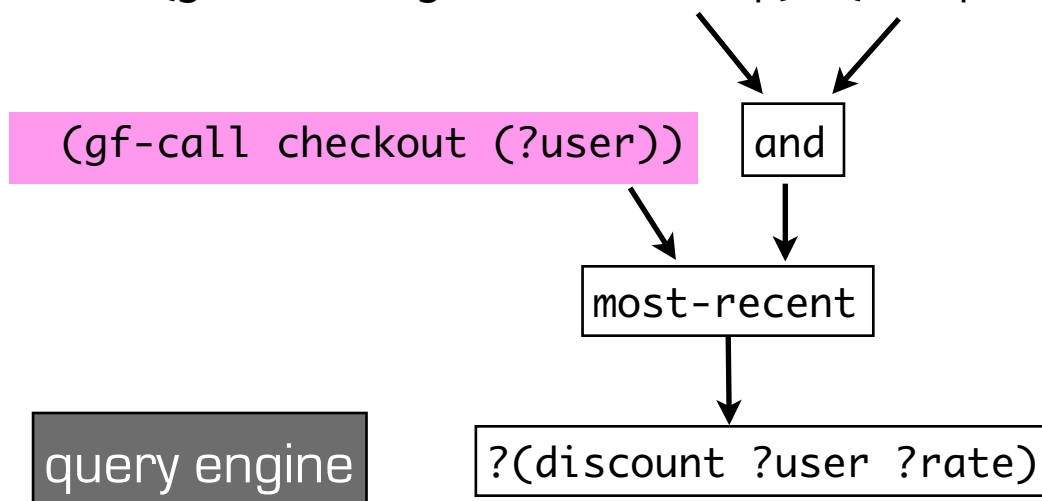
# Backward Chaining vs Rete Forward Chaining


"On every checkout, give a discount based on the rate at login time"

```
1. (set-rate *shop* 0.05)
2. (login *kris* *shop*)
   (buy *kris* *cd*)
   (set-rate *shop* 0.00)
   (checkout *kris*)
```

```
(1 gf-call 'set-rate <shop> 0.05)
(2 gf-call 'login <kris> <shop>)
```

```
(gf-call login ?user ?shop) (escape ?rate (current-rate ?shop))
```



↑ Backward Chaining 

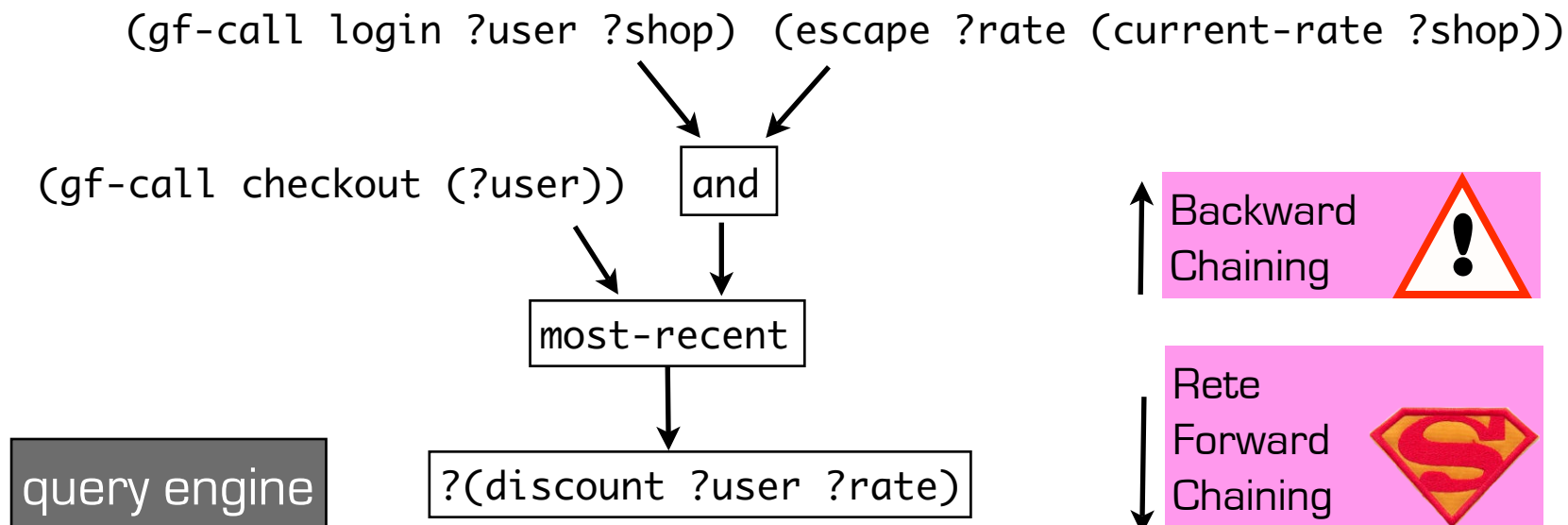
↓ Rete Forward Chaining 

# Backward Chaining vs Rete Forward Chaining

"On every checkout, give a discount  
based on the rate at login time"

```
1. (set-rate *shop* 0.05)
2. (login *kris* *shop*)
   (buy *kris* *cd*)
   (set-rate *shop* 0.00)
   (checkout *kris*)
```

```
(1 gf-call 'set-rate <shop> 0.05)
(2 gf-call 'login <kris> <shop>)
```

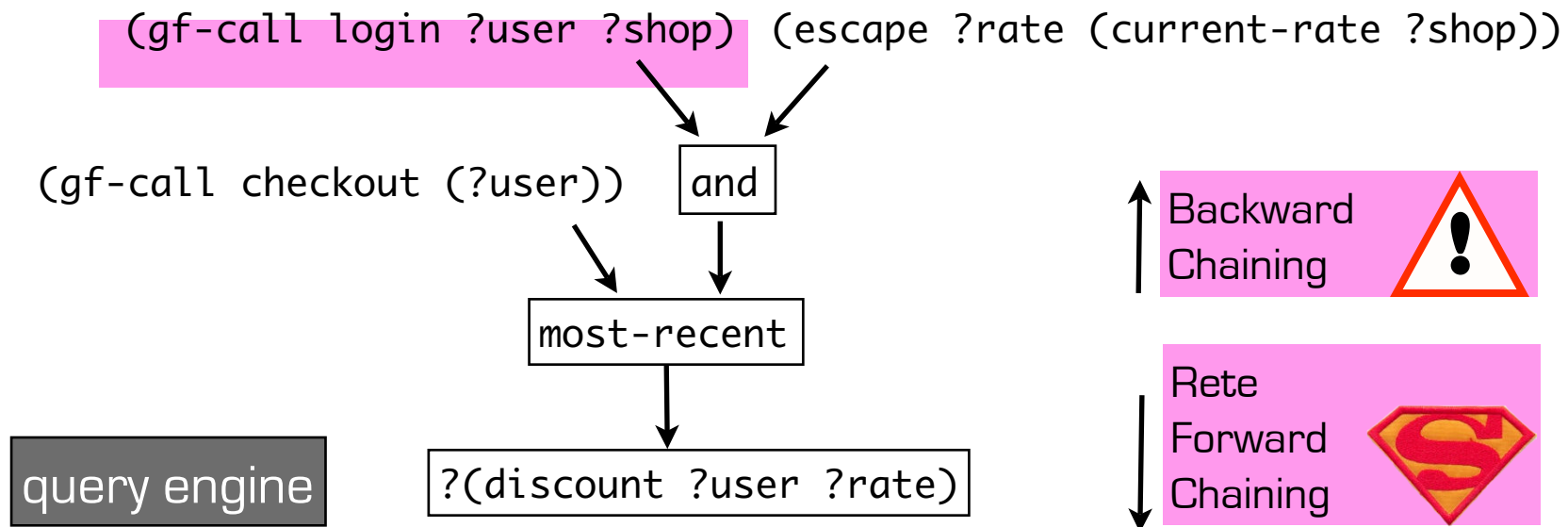


# Backward Chaining vs Rete Forward Chaining

"On every checkout, give a discount based on the rate at login time"

```
1. (set-rate *shop* 0.05)
2. (login *kris* *shop*)
   (buy *kris* *cd*)
   (set-rate *shop* 0.00)
   (checkout *kris*)
```

```
(1 gf-call 'set-rate <shop> 0.05)
(2 gf-call 'login <kris> <shop>)
```

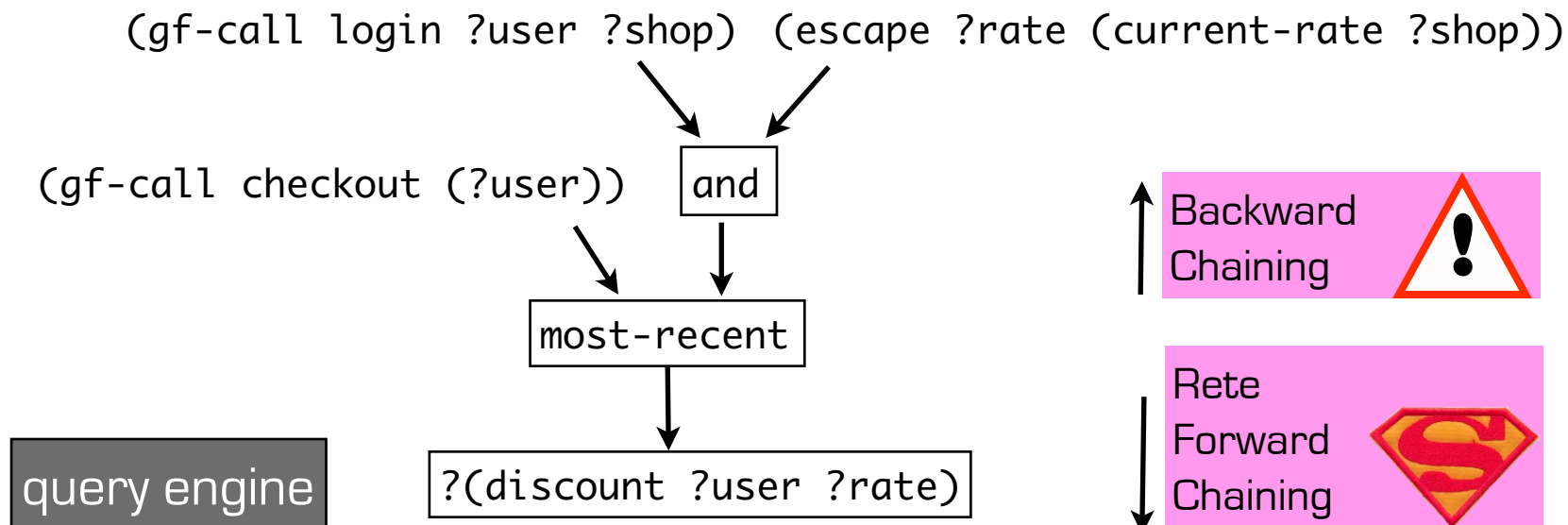


# Backward Chaining vs Rete Forward Chaining

"On every checkout, give a discount  
based on the rate at login time"

```
1. (set-rate *shop* 0.05)
2. (login *kris* *shop*)
   (buy *kris* *cd*)
   (set-rate *shop* 0.00)
   (checkout *kris*)
```

```
(1 gf-call 'set-rate <shop> 0.05)
(2 gf-call 'login <kris> <shop>)
```



# Backward Chaining vs Rete Forward Chaining

"On every checkout, give a discount  
based on the rate at login time"

```
1. (set-rate *shop* 0.05)
2. (login *kris* *shop*)
   (buy *kris* *cd*)
   (set-rate *shop* 0.00)
   (checkout *kris*)
```

```
(1 gf-call 'set-rate <shop> 0.05)
(2 gf-call 'login <kris> <shop>)
```

```
(2 gf-call 'login <kris> <shop>) (escape ?rate (current-rate ?shop))
```


(gf-call checkout (?user))

and

most-recent

query engine

?(discount ?user ?rate)

↑ Backward  
Chaining 

↓ Rete  
Forward  
Chaining 

# Backward Chaining vs Rete Forward Chaining

"On every checkout, give a discount  
based on the rate at login time"

```
1. (set-rate *shop* 0.05)
2. (login *kris* *shop*)
   (buy *kris* *cd*)
   (set-rate *shop* 0.00)
   (checkout *kris*)
```

```
(1 gf-call 'set-rate <shop> 0.05)
(2 gf-call 'login <kris> <shop>)
```

```
(2 gf-call 'login <kris> <shop>) (escape 0.05 (current-rate <shop>)))
```


(gf-call checkout (?user))

and

most-recent

query engine

?(discount ?user ?rate)

↑ Backward  
Chaining 

↓ Rete  
Forward  
Chaining 

# Backward Chaining vs Rete Forward Chaining

"On every checkout, give a discount  
based on the rate at login time"

```
1. (set-rate *shop* 0.05)
2. (login *kris* *shop*)
   (buy *kris* *cd*)
   (set-rate *shop* 0.00)
   (checkout *kris*)
```

```
(1 gf-call 'set-rate <shop> 0.05)
(2 gf-call 'login <kris> <shop>)
```

```
(2 gf-call 'login <kris> <shop>) (escape 0.05 (current-rate <shop>)))
```

(gf-call checkout (?user))

and

most-recent

query engine

?(discount ?user ?rate)

Backward  
Chaining



Rete  
Forward  
Chaining



# Backward Chaining vs Rete Forward Chaining

"On every checkout, give a discount based on the rate at login time"

```
1. (set-rate *shop* 0.05)
2. (login *kris* *shop*)
3. (buy *kris* *cd*)
   (set-rate *shop* 0.00)
   (checkout *kris*)
```

```
(1 gf-call 'set-rate <shop> 0.05)
(2 gf-call 'login <kris> <shop>)
```

```
(2 gf-call 'login <kris> <shop>) (escape 0.05 (current-rate <shop>)))
```


(gf-call checkout (?user))

and

most-recent

query engine

?(discount ?user ?rate)

↑ Backward Chaining 

↓ Rete Forward Chaining 



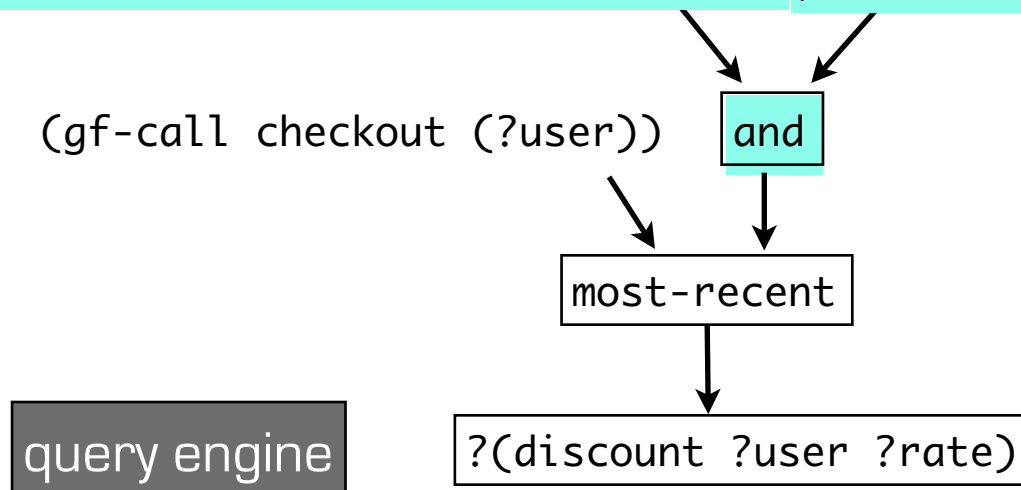
# Backward Chaining vs Rete Forward Chaining


"On every checkout, give a discount based on the rate at login time"

```
1. (set-rate *shop* 0.05)
2. (login *kris* *shop*)
3. (buy *kris* *cd*)
   (set-rate *shop* 0.00)
   (checkout *kris*)
```

```
(1 gf-call 'set-rate <shop> 0.05)
(2 gf-call 'login <kris> <shop>)
(3 gf-call 'buy <kris> <cd>)
```

```
(2 gf-call 'login <kris> <shop>) (escape 0.05 (current-rate <shop>))
```



Backward Chaining 

Rete Forward Chaining 

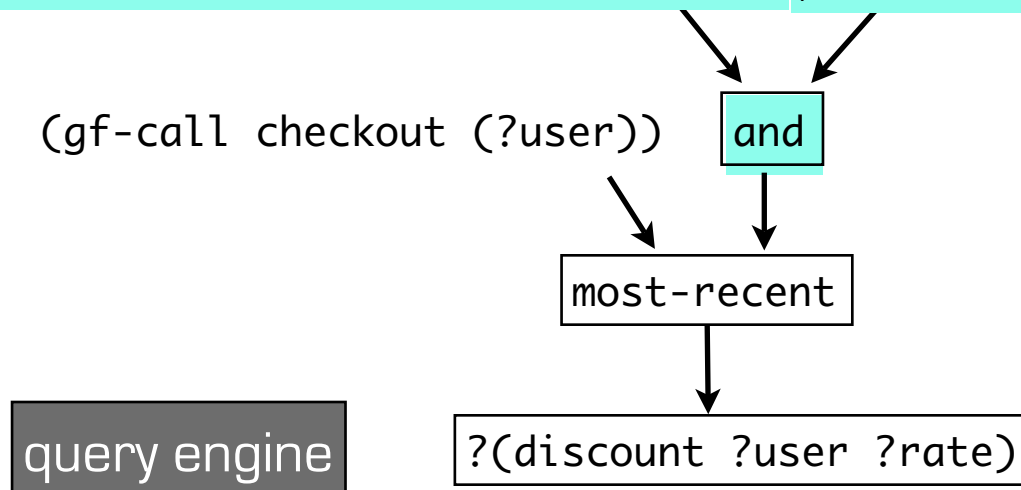
# Backward Chaining vs Rete Forward Chaining


"On every checkout, give a discount  
based on the rate at login time"

```
1. (set-rate *shop* 0.05)
2. (login *kris* *shop*)
3. (buy *kris* *cd*)
4. (set-rate *shop* 0.00)
   (checkout *kris*)
```

```
(1 gf-call 'set-rate <shop> 0.05)
(2 gf-call 'login <kris> <shop>)
(3 gf-call 'buy <kris> <cd>)
```

```
(2 gf-call 'login <kris> <shop>) (escape 0.05 (current-rate <shop>))
```



↑ Backward  
Chaining 

↓ Rete  
Forward  
Chaining 

# Backward Chaining vs Rete Forward Chaining

"On every checkout, give a discount based on the rate at login time"

```
1. (set-rate *shop* 0.05)
2. (login *kris* *shop*)
3. (buy *kris* *cd*)
4. (set-rate *shop* 0.00)
   (checkout *kris*)
```

```
(1 gf-call 'set-rate <shop> 0.05)
(2 gf-call 'login <kris> <shop>)
(3 gf-call 'buy <kris> <cd>)
(4 gf-call 'set-rate <shop> 0.00)
```

```
(2 gf-call 'login <kris> <shop>) (escape 0.05 (current-rate <shop>)))
```


(gf-call checkout (?user))

and

most-recent

query engine

?(discount ?user ?rate)

Backward  
Chaining 

Rete  
Forward  
Chaining 

# Backward Chaining vs Rete Forward Chaining

"On every checkout, give a discount  
based on the rate at login time"

```
1. (set-rate *shop* 0.05)
2. (login *kris* *shop*)
3. (buy *kris* *cd*)
4. (set-rate *shop* 0.00)
5. (checkout *kris*)
```

```
(1 gf-call 'set-rate <shop> 0.05)
(2 gf-call 'login <kris> <shop>)
(3 gf-call 'buy <kris> <cd>)
(4 gf-call 'set-rate <shop> 0.00)
```

```
(2 gf-call 'login <kris> <shop>) (escape 0.05 (current-rate <shop>))
```


(gf-call checkout (?user))

and

most-recent

query engine

?(discount ?user ?rate)

Backward  
Chaining 

Rete  
Forward  
Chaining 

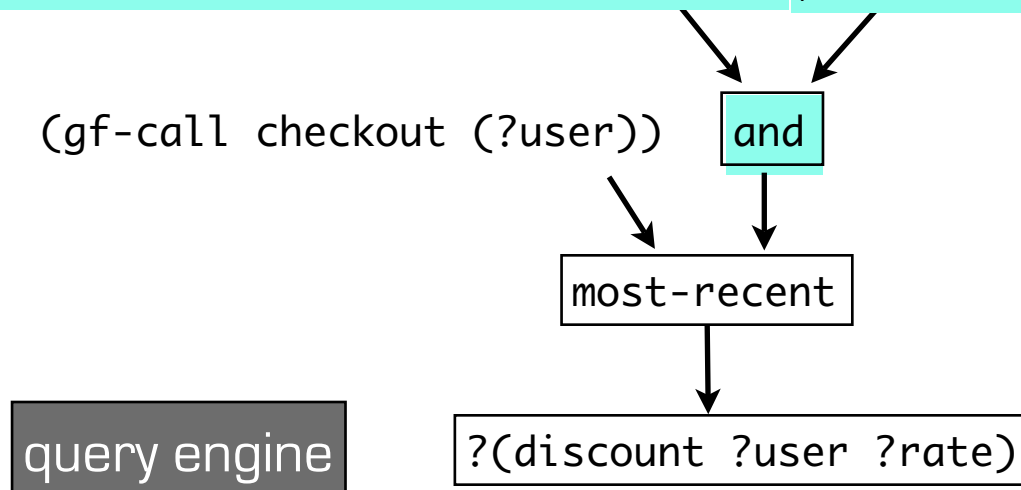
# Backward Chaining vs Rete Forward Chaining


"On every checkout, give a discount based on the rate at login time"

```
1. (set-rate *shop* 0.05)
2. (login *kris* *shop*)
3. (buy *kris* *cd*)
4. (set-rate *shop* 0.00)
5. (checkout *kris*)
```

```
(1 gf-call 'set-rate <shop> 0.05)
(2 gf-call 'login <kris> <shop>)
(3 gf-call 'buy <kris> <cd>)
(4 gf-call 'set-rate <shop> 0.00)
(5 gf-call 'checkout <kris>)
```

```
(2 gf-call 'login <kris> <shop>) (escape 0.05 (current-rate <shop>)))
```



Backward Chaining 

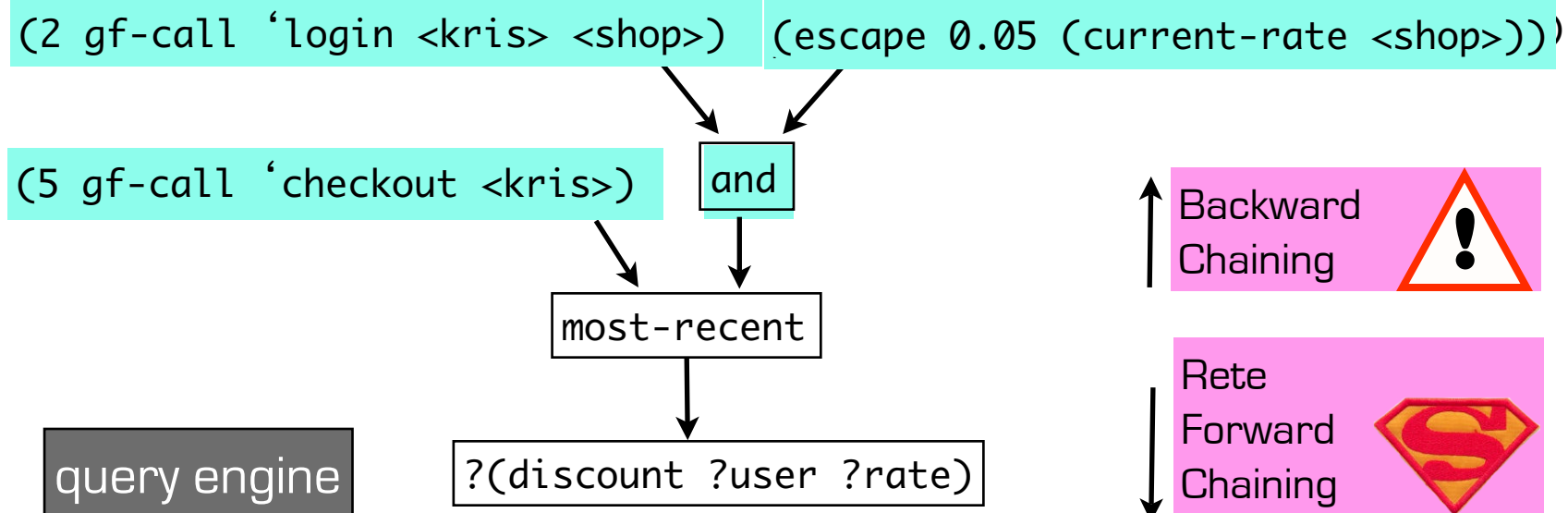
Rete Forward Chaining 

# Backward Chaining vs Rete Forward Chaining

"On every checkout, give a discount  
based on the rate at login time"

```
1. (set-rate *shop* 0.05)
2. (login *kris* *shop*)
3. (buy *kris* *cd*)
4. (set-rate *shop* 0.00)
5. (checkout *kris*)
```

```
(1 gf-call 'set-rate <shop> 0.05)
(2 gf-call 'login <kris> <shop>)
(3 gf-call 'buy <kris> <cd>)
(4 gf-call 'set-rate <shop> 0.00)
(5 gf-call 'checkout <kris>)
```

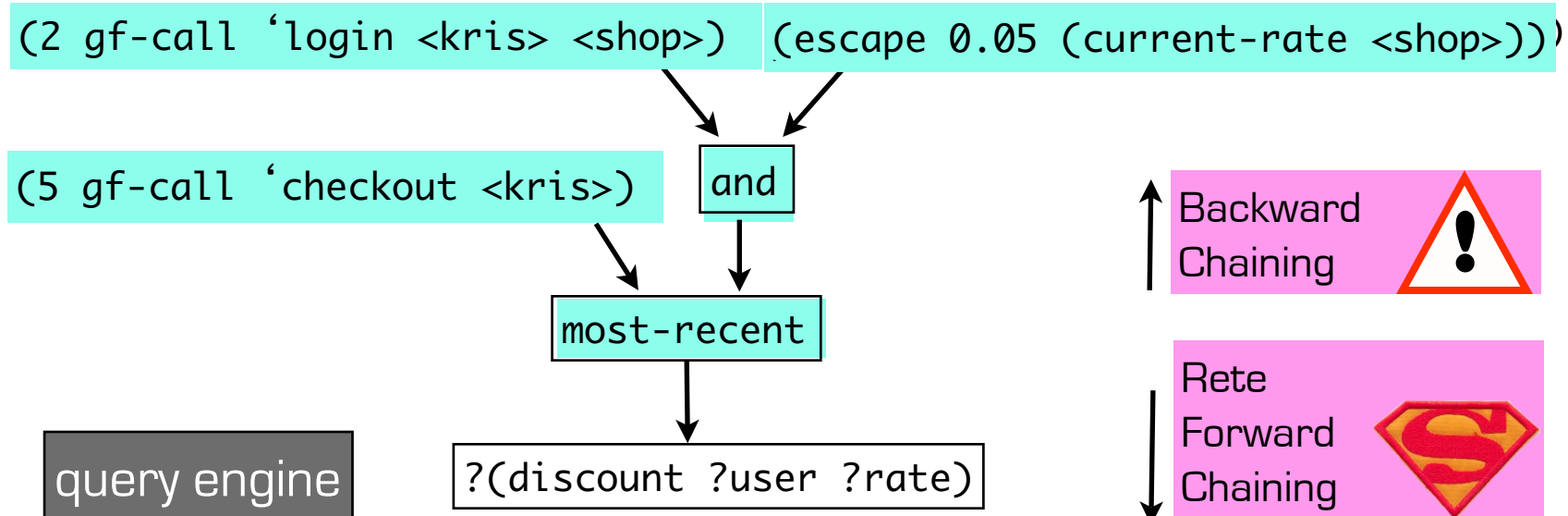


# Backward Chaining vs Rete Forward Chaining

"On every checkout, give a discount based on the rate at login time"

```
1. (set-rate *shop* 0.05)
2. (login *kris* *shop*)
3. (buy *kris* *cd*)
4. (set-rate *shop* 0.00)
5. (checkout *kris*)
```

```
(1 gf-call 'set-rate <shop> 0.05)
(2 gf-call 'login <kris> <shop>)
(3 gf-call 'buy <kris> <cd>)
(4 gf-call 'set-rate <shop> 0.00)
(5 gf-call 'checkout <kris>)
```

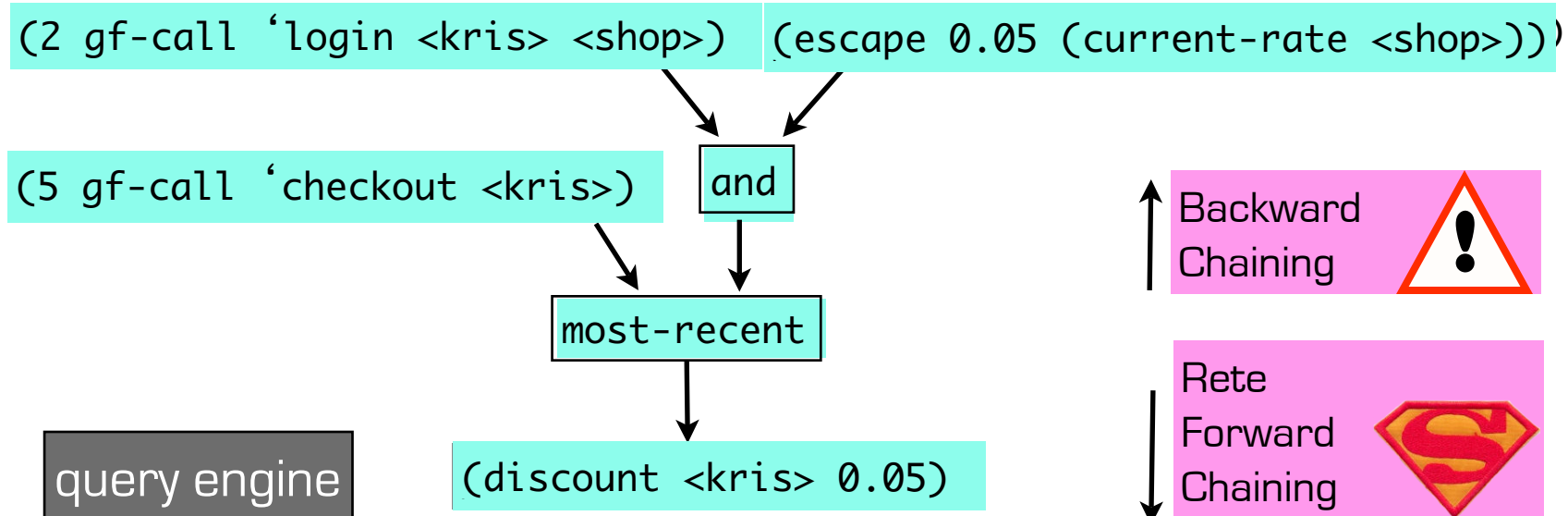


# Backward Chaining vs Rete Forward Chaining

"On every checkout, give a discount based on the rate at login time"

```
1. (set-rate *shop* 0.05)
2. (login *kris* *shop*)
3. (buy *kris* *cd*)
4. (set-rate *shop* 0.00)
5. (checkout *kris*)
```

```
(1 gf-call 'set-rate <shop> 0.05)
(2 gf-call 'login <kris> <shop>)
(3 gf-call 'buy <kris> <cd>)
(4 gf-call 'set-rate <shop> 0.00)
(5 gf-call 'checkout <kris>)
```



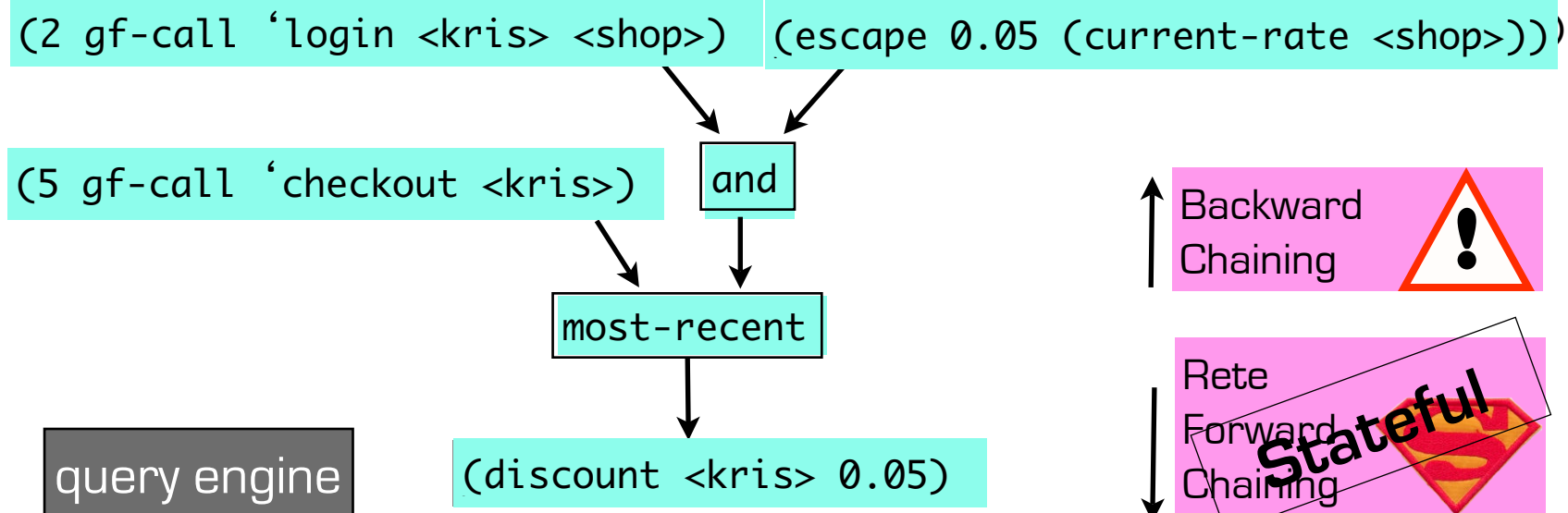


# Backward Chaining vs Rete Forward Chaining

"On every checkout, give a discount  
based on the rate at login time"

```
1. (set-rate *shop* 0.05)
2. (login *kris* *shop*)
3. (buy *kris* *cd*)
4. (set-rate *shop* 0.00)
5. (checkout *kris*)
```

```
(1 gf-call 'set-rate <shop> 0.05)
(2 gf-call 'login <kris> <shop>)
(3 gf-call 'buy <kris> <cd>)
(4 gf-call 'set-rate <shop> 0.00)
(5 gf-call 'checkout <kris>)
```

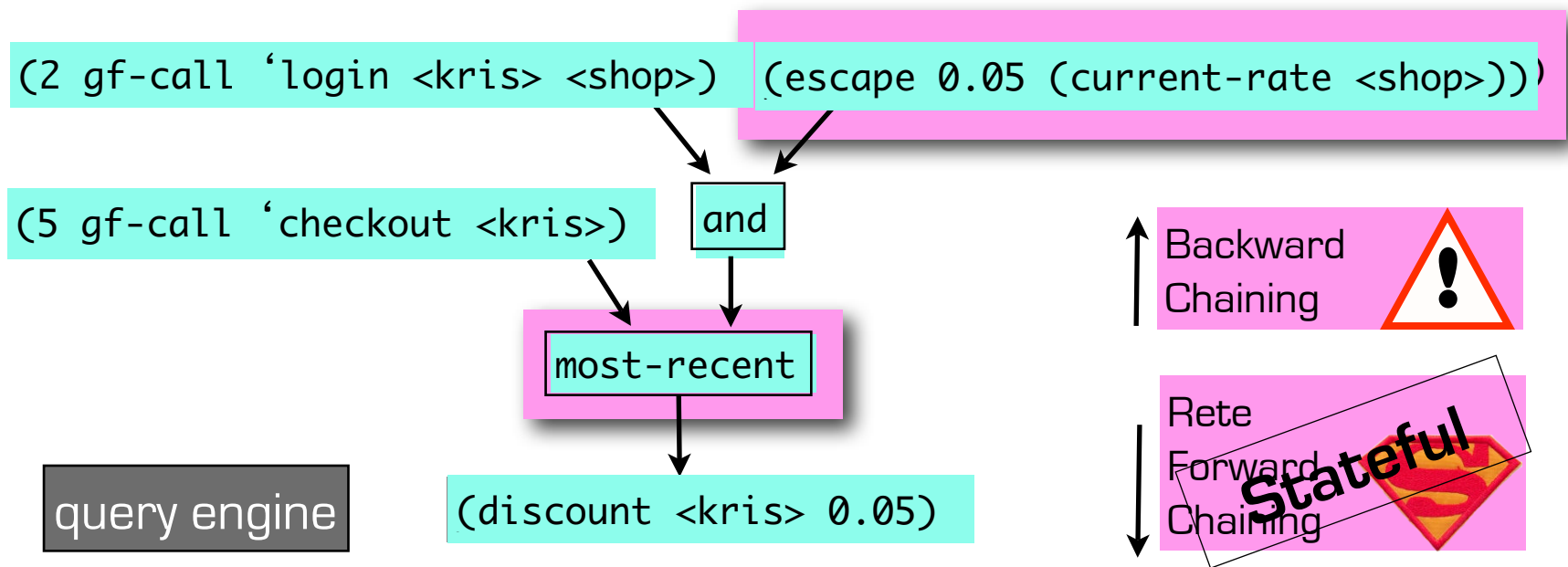


# Backward Chaining vs Rete Forward Chaining

"On every checkout, give a discount  
based on the rate at login time"

```
1. (set-rate *shop* 0.05)
2. (login *kris* *shop*)
3. (buy *kris* *cd*)
4. (set-rate *shop* 0.00)
5. (checkout *kris*)
```

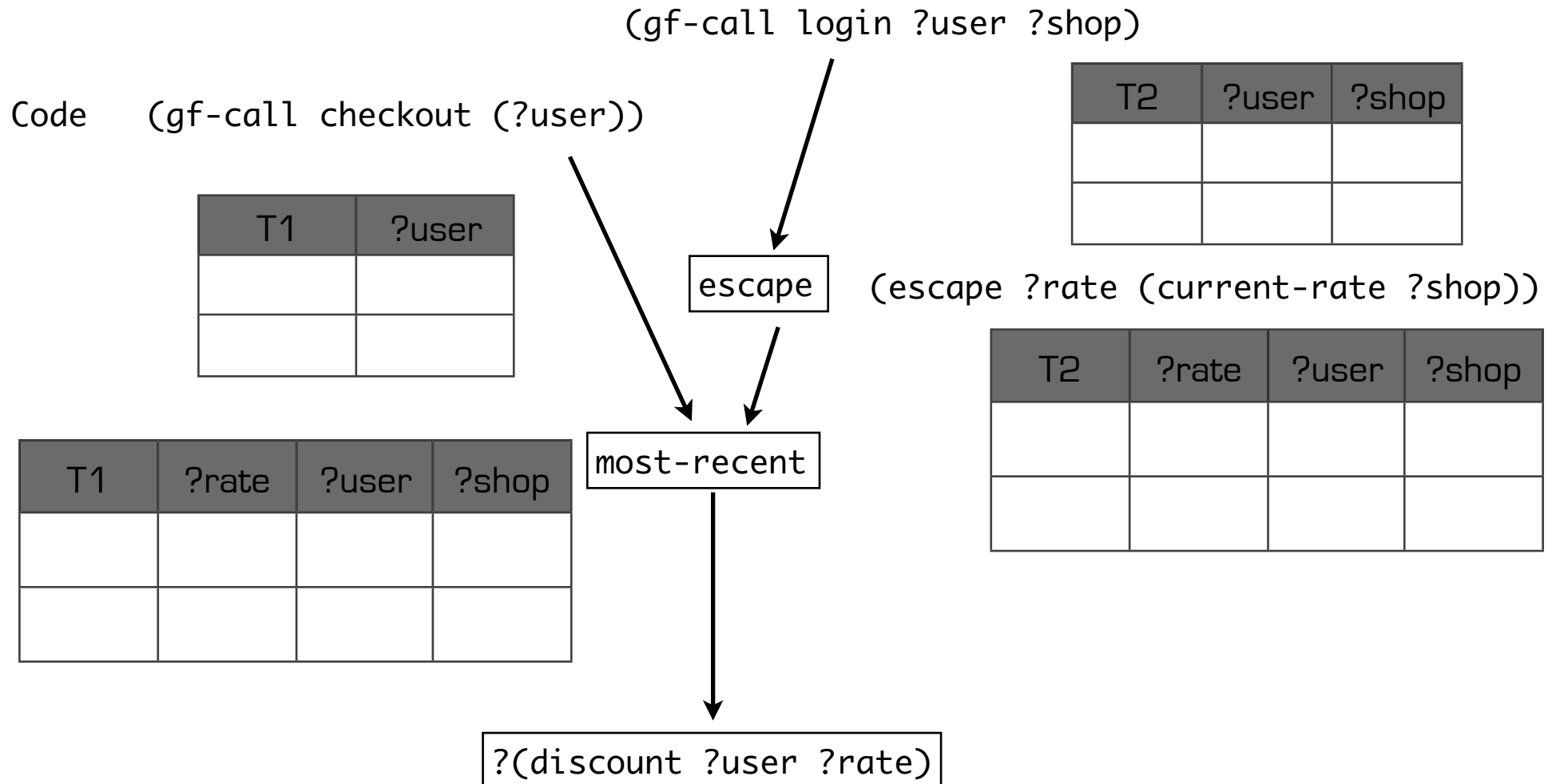
```
(1 gf-call 'set-rate <shop> 0.05)
(2 gf-call 'login <kris> <shop>)
(3 gf-call 'buy <kris> <cd>)
(4 gf-call 'set-rate <shop> 0.00)
(5 gf-call 'checkout <kris>)
```



# Reducing Memory Overhead

```
(at ((gf-call checkout (?user))  
    (most-recent (gf-call login ?user ?shop)  
                  (escape ?rate (current-rate ?shop)))))  
(discount ?user ?rate))
```

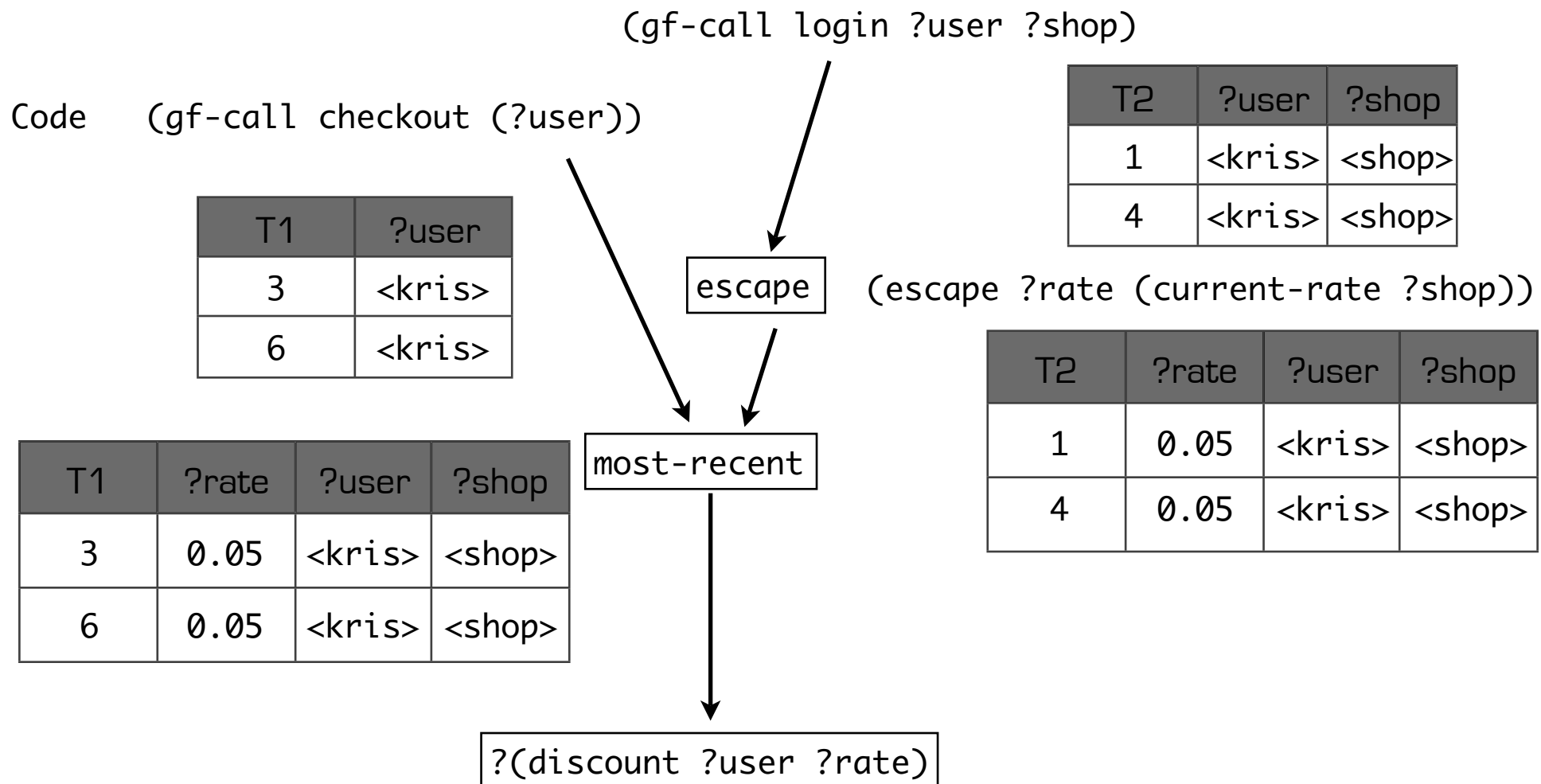
```
(login *kris* *shop*)  
(buy *kris* *cd*)  
(checkout *kris*)  
(login *kris* *shop*)  
(buy *kris* *dvd*)  
(checkout *kris*)
```



# Reducing Memory Overhead

```
(at ((gf-call checkout (?user))
      (most-recent (gf-call login ?user ?shop)
                    (escape ?rate (current-rate ?shop)))))
(discount ?user ?rate))
```

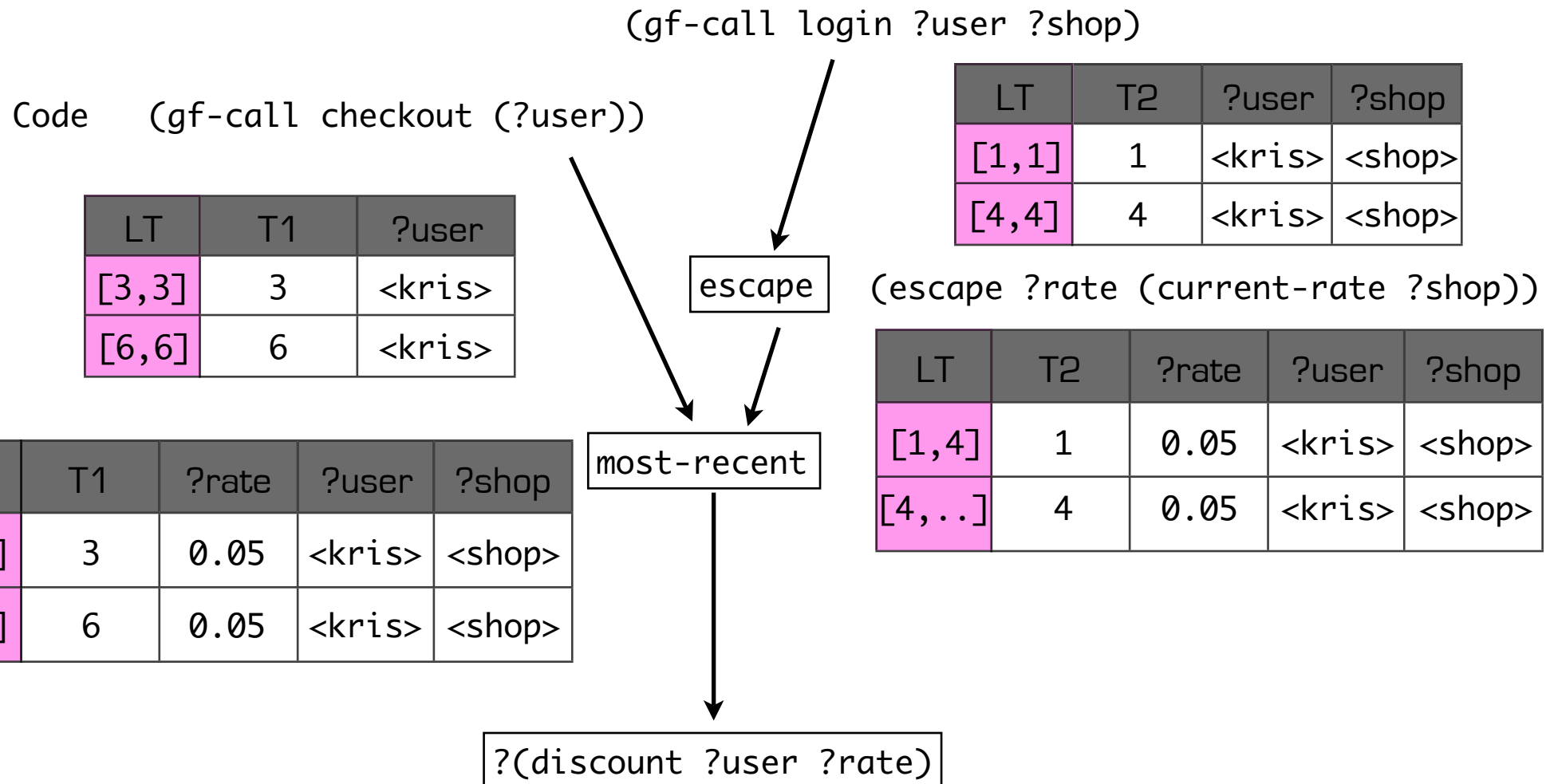
```
(login *kris* *shop*)
(buy *kris* *cd*)
(checkout *kris*)
(login *kris* *shop*)
(buy *kris* *dvd*)
(checkout *kris*)
```



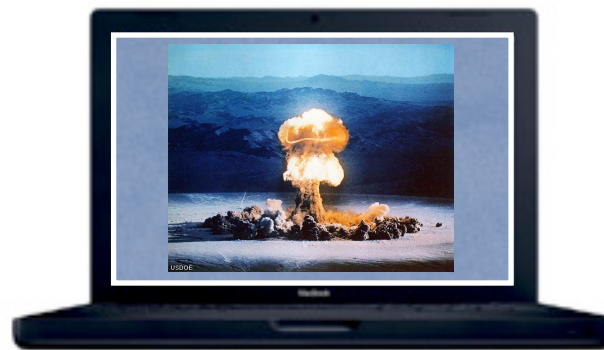
# Reducing Memory Overhead

```
(at ((gf-call checkout (?user))
      (most-recent (gf-call login ?user ?shop)
                    (escape ?rate (current-rate ?shop)))))
(discount ?user ?rate))
```

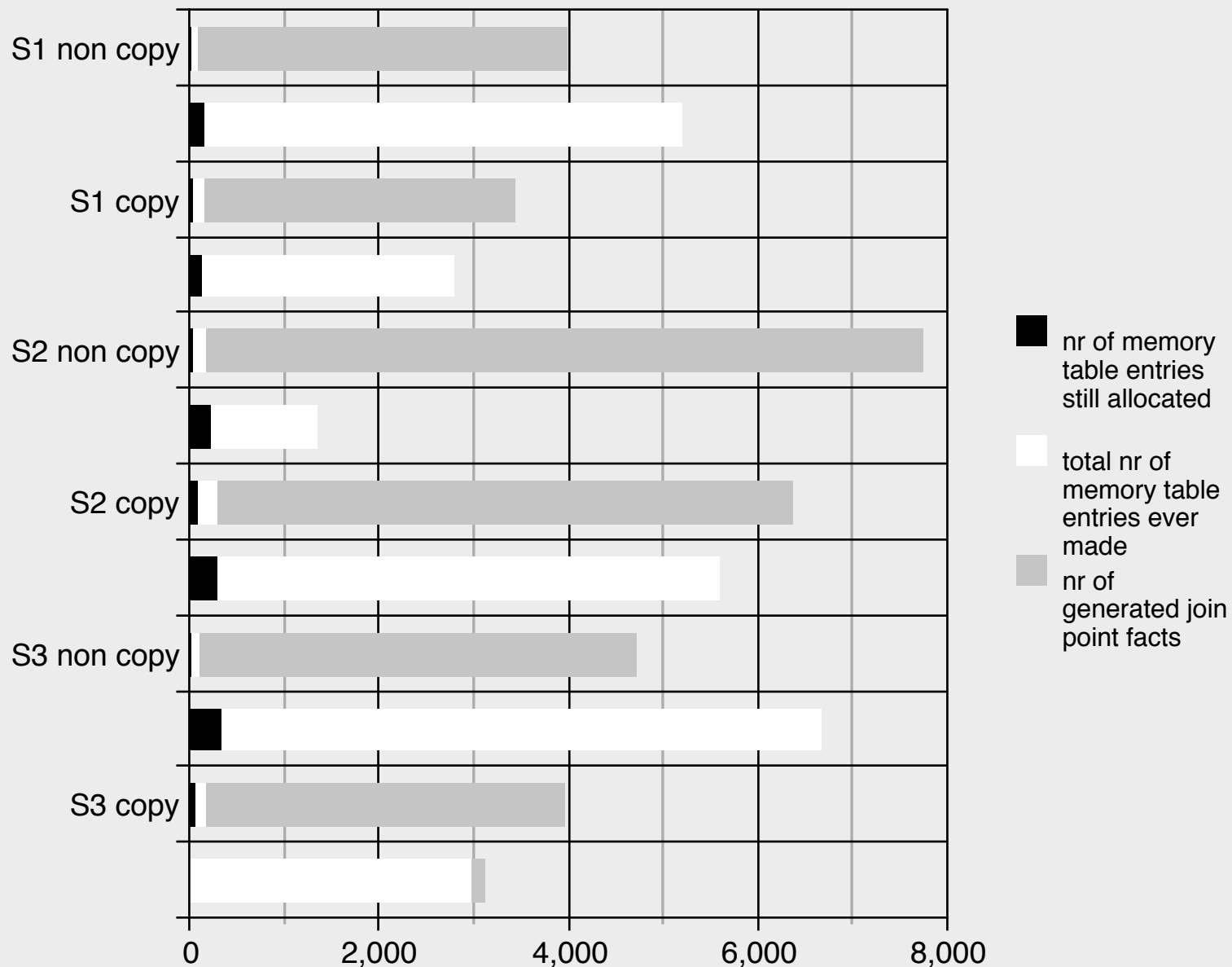
```
(login *kris* *shop*)
(buy *kris* *cd*)
(checkout *kris*)
(login *kris* *shop*)
(buy *kris* *dvd*)
(checkout *kris*)
```



# DEMO



# Reducing Memory Overhead (2)



## Modularizing crosscuts in an e-commerce application in Lisp using HALO

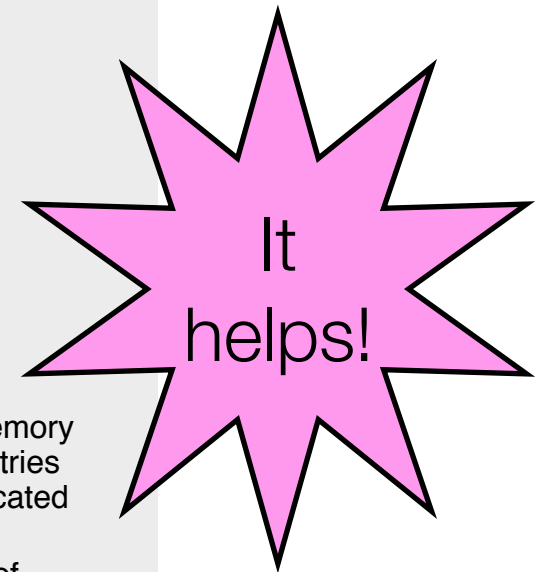
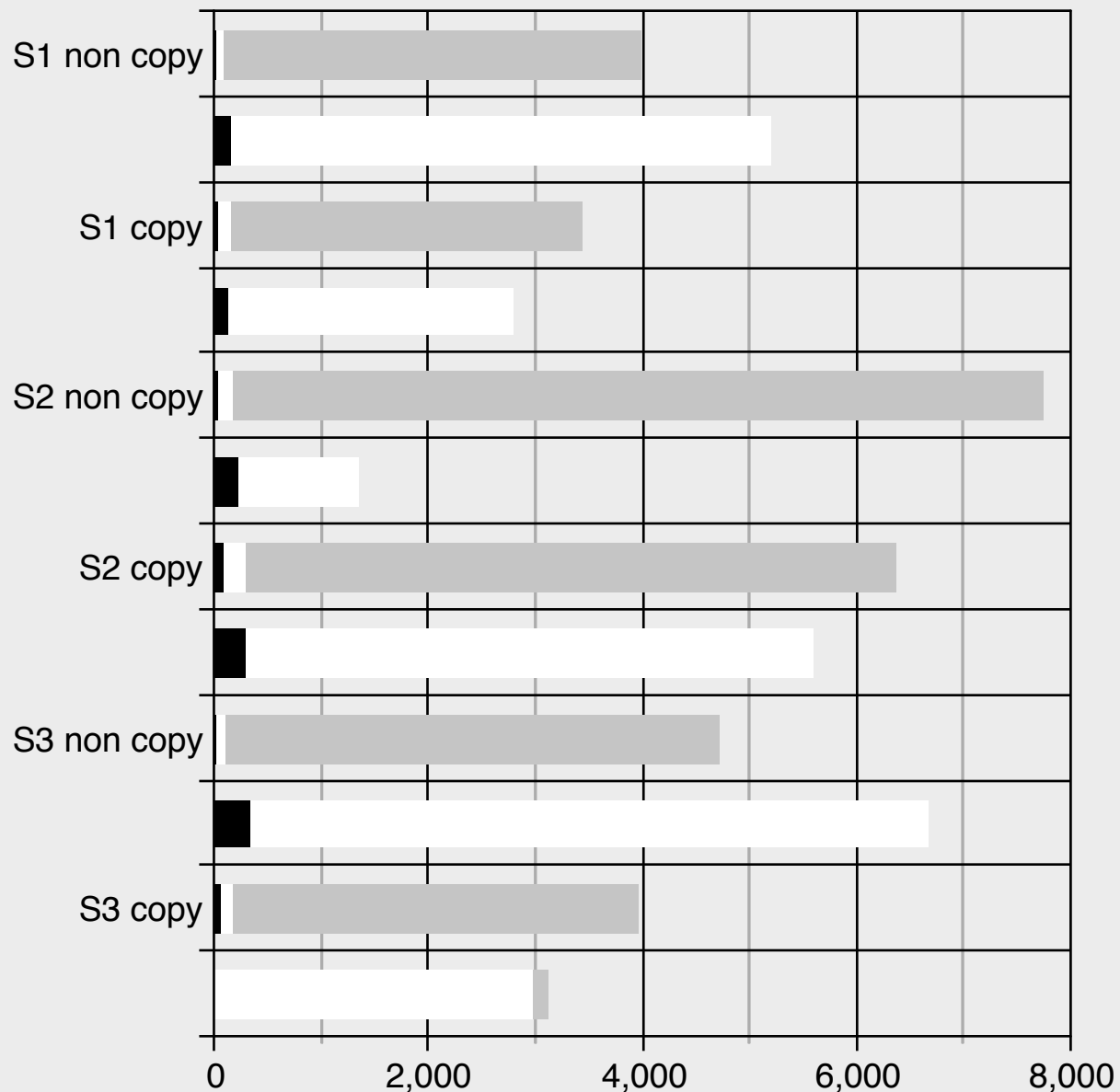
Charlotte Herzeel, Kris Gybels, Pascal Costanza, Theo D'Hondt  
In "Proc. of the International Lisp Conference", 2007



## Escaping with future variables in HALO

Charlotte Herzeel, Kris Gybels, Pascal Costanza  
In "Proc. of the Runtime Verification Workshop", 2007

# Reducing Memory Overhead (2)



- nr of memory table entries still allocated
- total nr of memory table entries ever made
- nr of generated join point facts



**Modularizing crosscuts in an e-commerce application in Lisp using HALO**  
Charlotte Herzeel, Kris Gybels, Pascal Costanza, Theo D'Hondt  
In "Proc. of the International Lisp Conference", 2007



**Escaping with future variables in HALO**  
Charlotte Herzeel, Kris Gybels, Pascal Costanza  
In "Proc. of the Runtime Verification Workshop", 2007



# Summary & Future Work

- Summary:
  - history-based AOP for CLOS ; stateful pointcuts
  - temporal logic-based: abstraction & logic variables
  - Rete forward chaining-based implementation
  - memory reduction
- Future Work:
  - language design:
    - exploit logic-programming (i.e. unification & recursion)
    - exploit temporal logic-programming (mixing temporal logic/advice code)
    - before/ after ... what about around? (roll-back mechanism)
  - reducing memory overhead:
    - shadow weaving

# Summary & Future Work

- Summary:
  - history-based AOP for CLOS ; stateful pointcuts
  - temporal logic-based: abstraction & logic variables
  - Rete forward chaining-based implementation
  - memory reduction
- Future Work:
  - language design:
    - exploit logic-programming (i.e. unification & recursion)
    - exploit temporal logic-programming (mixing temporal logic/advice code)
    - before/ after ... what about around? (roll-back mechanism)
  - reducing memory overhead:
    - shadow weaving

Thanks for listening!  
Questions?

# Summary & Future Work

- Summary:
  - history-based AOP for CLOS ; stateful pointcuts
  - temporal logic-based: abstraction & logic variables
  - Rete forward chaining-based implementation
  - memory reduction
- Future Work:
  - language design:
    - exploit logic-programming (i.e. unification & recursion)
    - exploit temporal logic-programming (mixing temporal logic/advice code)
    - before/after ... what about around? (roll-back mechanism)
  - reducing memory overhead:
    - shadow weaving

Thanks for listening!

Questions?

<http://prog.vub.ac.be/HALO>

# References

- Supporting Software Development Through Declaratively Codified Programming Patterns, Kim Mens, Isabel Michiels, Roel Wuyts In "Journal on Expert Systems with Applications, Elsevier Publications, pages 405-413, number 4, Volume 23, November", 2002
- **Arranging language features for more robust pattern-based crosscuts**  
Kris Gybels, Johan Brichau, In "Proceedings of the Second International Conference on Aspect-Oriented Software Development", 2003
- Seamless Integration of Rule-Based Knowledge and Object-Oriented Functionality with Linguistic Symbiosis, Maja D'Hondt, Kris Gybels, Viviane Jonckers, In "Proceedings of the 19th Annual ACM Symposium on Applied Computing (SAC 2004), Special Track on Object-Oriented Programming, Languages and Systems. Nicosia, Cyprus. ACM. March 2004.", 2004
- **Context-aware Aspects**, Eric Tanter, Kris Gybels, Marcus Denker, Alexandre Bergel, Proceedings of the 5th International Symposium on Software Composition (SC 2006), LNCS 4089, pp.227-249, Springer-Verlag, March 2006, Vienna, Austria
- **Expressive Pointcuts for Increased Modularity**, Klaus Ostermann and Mira Mezini and Christoph Bockisch, Proceedings of the European Conference on Object-Oriented Programming (ECOOP)
- **Modularizing Crosscuts in an E-commerce Application in Lisp using HALO** , Charlotte Herzeel, Kris Gybels, Pascal Costanza, Theo D'Hondt , International Lisp Conference 2007
- **Escaping with future variables in HALO** , Charlotte Herzeel, Kris Gybels, Pascal Costanza, Proceedings of the Runtime Verification Workshop 2007

