Prolog

Prof. Dr. Stefan Böttcher
Fakultät EIM, Institut für Informatik
Universität Paderborn
SS 2019

Contents:
- Introduction: Prolog as a database language
- List programming and 1:1 machine translation
- Puzzles, quizes, games
- Inference engines, meta-interpreters, ...
- Parsers and interpreters for grep, XML, SQL, German, English
- Compilers, translators, natural language understanding,
- Question answering systems

Prerequisites and requirements

1. Prolog programming assignments
   - given each Tuesday directly in or after the lecture
   - have to be solved individually by each student during the next six days,
   - solutions have to be presented and explained on Monday (6 days after the lecture) within one of the exercise groups

2. Presenting your solutions within the exercise times is mandatory to pass the exam.

Exercises: to be done at home – starting today !
Presentation times:
Mo. 9:15-10:45 , Mo. 10:50-12:20, Mo 12:25-13:55

Required previous knowledge

Programming language Prolog, and of Relational Algebra,
exactly of the amount provided in the course "Grundlagen Datenbanken".

If you did not join the course "Grundlagen Datenbanken"
or forgot the Prolog part of it, you should read and work through
the following material, **before** doing the first exercises:

a) Read about Selection, Projection, Union, Set Difference, Intersection, Join, Cartesian Product and Division in any good text book on database systems, e.g.: Hector Garcia Molina, Jeffrey, D. Ullman, and Jenifer Widom: Database Systems. The Complete Book. Prentice Hall 2008, pp 189-224,302-310, and 463-480.

b) Watch the following video about the first steps to Prolog: Programming in Prolog: this is The Simple Engineer's four part video introductionusing SWI-Prolog.
https://www.youtube.com/watch?v=gJOZZvYijqk&list=PLVmRRBrc2pRCWtYk752jCIfhD8GmoYfc_
This is a nice small video sequence to start with which covers parts of the first two lectures.
It is definitely less challenging than our course.
As we use SWI-Prolog throughout the lecture, this video is recommended as first video about Prolog.

c) Derek Banas's Prolog Tutorial. https://www.youtube.com/watch?v=SykxWpFwMGs .
This is an hour-long video tutorial, which is based on GNU Prolog (=gprolog) and requires an installation of C++. Please use SWI-Prolog instead. You could skip the first minutes and start at minute 5:15, and install and use **SWI-Prolog 8.0.2-1** instead.

d) Mike Brayshaw: http://www.doc.gold.ac.uk/~mas02gw/prolog_tutorial/prologpages/
A very basic intro into Prolog (covering at most the first two or three lectures).

e) Bernardo Pires: Try Logic Programming! A Gentle Introduction to Prolog.
Another very basic introduction to Prolog (covering the first two or three lectures)

f) Marc Bezem: A Prolog Compendium (pdf) www.ii.uib.no/~bezem/Prolog_Tutorial.pdf
Useful as introduction for the first two or three weeks of our course (or so).

Getting started with Prolog

Try to install SWI-Prolog **8.0.2-1** on your computer alone!
→ Installation instructions on web page
→ Reserve enough time !
→ Start today !

If you could not install SWI-Prolog **8.0.2-1** alone
last support for getting started with Prolog is
Wednesday 10.4.2019 9:15 – 12:45
→ save the date (in case you did not succeed before)
→ if you installed everything alone successfully,
do not come to this specific installation date

Why Prolog for AI?

1. Declarative programming
   → ”say what you want, and Prolog does it for you”

2. Tree data structures and tree unification
   → example given on white board

Our first example database

student

sID	firstname	surname	term
1000 ,	‘Anna‘ ,	‘Arm‘ ,	‘ti2‘
1001 ,	‘Rita‘ ,	‘Reich‘ ,	‘ti2‘
1002 ,	‘Peter‘ ,	‘Reich‘ ,	‘ti2‘
1003 ,	‘Peter‘ ,	‘Petersen‘ ,	‘ti7‘

← tuple
=data record

course

term	subject
‘ti2‘ ,	‘Mathe2‘
‘ti2‘ ,	‘Physics2‘
‘ti7‘ ,	‘pdv7‘

Prolog as a database language - idea

student

constant

	sID	firstname	surname	term	
student(	1000 ,	‘Anna‘ ,	‘Arm‘ ,	‘ti2‘	) . ←tuple **= fact**
student(	1001 ,	‘Rita‘ ,	‘Reich‘ ,	‘ti2‘	) . =data record
student(	1002 ,	‘Peter‘ ,	‘Reich‘ ,	‘ti2‘	) .
student(	1003 ,	‘Peter‘ ,	‘Petersen‘ ,	‘ti7‘	) .

course

	term	subject	
course(	‘ti2‘ ,	‘Mathe2‘	) .
course(	‘ti2‘ ,	‘Physics2‘	) .
course(	‘ti7‘ ,	‘pdv7‘	) .

Prolog as a database language - relation

predicate
= relation
= procedure

```
student(  1000  ,  'Anna'   , 'Arm' ,        'ti2'  ) .   ← tuple  = fact
student(  1001  ,  'Rita'   , 'Reich' ,      'ti2'  ) .   =data record
student(  1002  ,  'Peter'  , 'Reich' ,      'ti2'  ) .
student(  1003  ,  'Peter'  , 'Petersen' ,   'ti7'  ) .

course(   'ti2'   ,  'Mathe2'    ) .
course(   'ti2'   ,  'Physics2'  ) .
course(   'ti7'   ,  'pdv7'.     ) .
```

Prolog as database language - syntax

predicate
= relation
= procedure

constant → integer, atom

atom: *starts with lower case or is enclosed in ‘ ‘*

```
student(  1000  ,  ‘Anna‘   , ‘Arm‘ ,         ‘ti2‘  ) .
student(  1001  ,  ‘Rita‘   , ‘Reich‘ ,       ‘ti2‘  ) .
student(  1002  ,  ‘Peter‘  , ‘Reich‘ ,       ‘ti2‘  ) .
student(  1003  ,  ‘Peter‘  , ‘Petersen‘ ,    ‘ti7‘  ) .
```

tuple=fact
=data record

no blank !

goal = sub-query

```
?-  student( M , F , _ , _ )  .
```

variable   anonymous variable

Answer generation by variable binding

constant

integer atom

```
    student( 1000 , 'Anna' , 'Arm' ,          'ti2' ) .
?-  student(  M   , 'Peter' ,  _   ,           _    ) .
    student( 1002 , 'Peter' , 'Reich' ,        'ti2' ) .
    student( 1003 , 'Peter' , 'Petersen' ,     'ti7' ) .
```

unification fails

unification binds: M=1002

...

anonymous variables _ and _ can be bound differently !

Select-Project-Join-Queries

**Query: in which term S is Anna Arm,
and which courses C everyone must take in term S?**

goal = sub-query   sub-query

?- student(_, ‘Anna‘ , ‘Arm‘ , S) , course(S , C) .

projection, selection   join

Join and cartesian product

Query: who (is in which term S and) has to take (therefore) which courses C?

goal = sub-query   sub-query

```
?-    student( M, F , N , S )    ,    course( S , C )  .
```

Join

Query: considering students and offered courses who can take which courses?

```
?-    student( M, F , N , S )    ,    course( S2 , C )  .
```

cartesian product

Prolog rules - syntax

rule: head = view :- goals

goals: goal = sub-query, sub-query

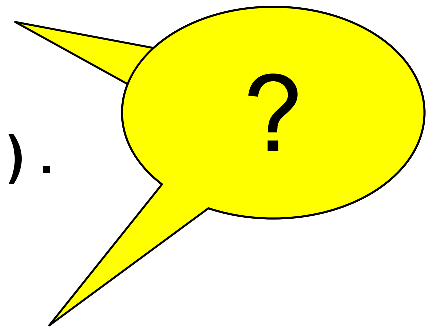
```
mustTake( M, F, N, S, C )  :-  student( M, F, N, S ) , course( S, C ) .
```

- head = view: `mustTake( M, F, N, S, C )`
- goal = sub-query: `student( M, F, N, S )`
- sub-query: `course( S, C )`

```
?- mustTake(1000 , _ , _ , _ , C )  .
```

query

What are the answers to these queries?



1. ?- student(_ , ‘Anna‘ , N , S) .
2. ?- student(_ , ‘Anna‘ , S , S) .
3. ?- course(S , C) , student(M , F , ‘Petersen‘ , S) .
4. ?- mustPeter(M , _ , _ , _ , C) .
5. ?- mustPeter(_ , _ , N , _ , ‘ti2‘) .
6. ?- mustPeter(_ , F , N1 , _ , S) , mustPeter(_ , F , N2 , _ , S) .

```
mustPeter( M, F, N, S, C )  :-   student( M, ‘Peter‘, N, S ) , course( S, C )  .

student(  1000  ,  ‘Anna‘   , ‘Arm‘ ,         ‘ti2‘  ) .
student(  1001  ,  ‘Rita‘   , ‘Reich‘ ,       ‘ti2‘  ) .
student(  1002  ,  ‘Peter‘  , ‘Reich‘ ,       ‘ti2‘  ) .
student(  1003  ,  ‘Peter‘  , ‘Petersen‘ , ‘ti7‘  ) .

course(   ‘ti2‘    ,  ‘Mathe2‘    ) .
course(   ‘ti2‘    ,  ‘Physics2‘ ).
course(   ‘ti7‘    , ‘pdv7‘.        ) .
```

same as before
→ look at your slides
printed on paper !

declarative semantics

Prolog:

mustTake(M, F, N, S, C) :- student(M, F, N, S) , course(S, C) .

predicate calculus

mustTake(M, F, N, S, C) $\Leftarrow$ student(M, F, N, S) $\wedge$ course(S, C) .

if and

relational algebra

mustTake(M, F, N, S, C) := student |X| course

$4 = 1$

SQL

create view

mustTake as select * from student ST, course CO

where ST . S = CO . S

procedural semantics: data flow

rule:

head = view | **goal = sub-query** | **sub-query**

```
mustTake( M, F, N, S, C )  :-  student( M, F, N, S ) , course( S, C )  .
```

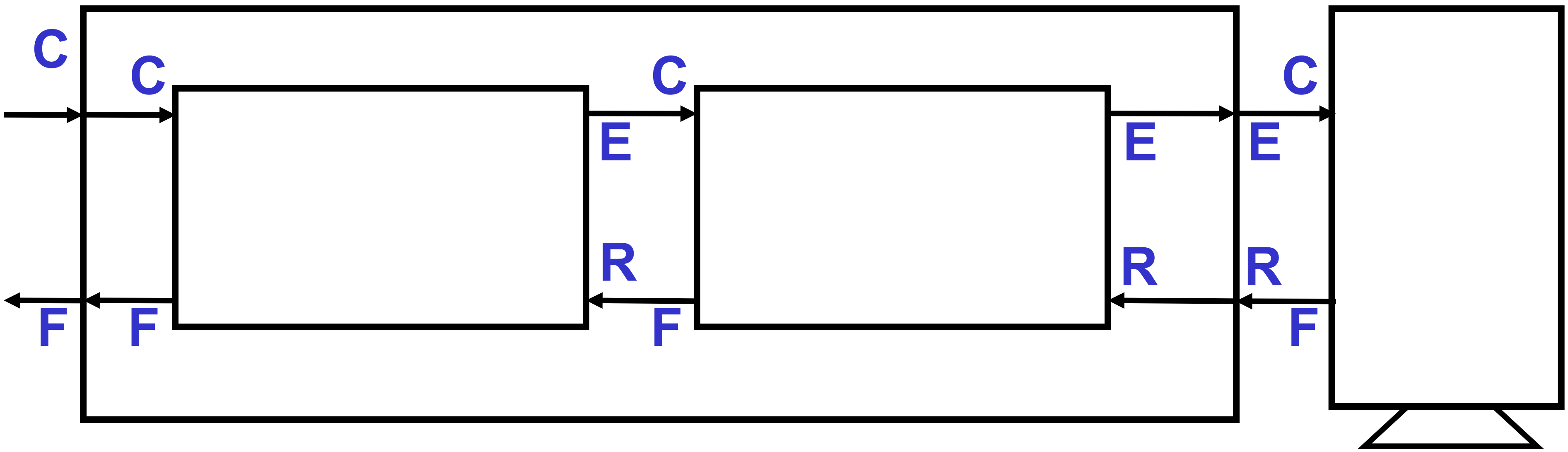
```
?- mustTake( 1000 , _, _, _, C )  .
```

⇒ **variable bindings for input and output parameters**

→ **transport of variable bindings inside a rule**

procedural semantics: 4-port model

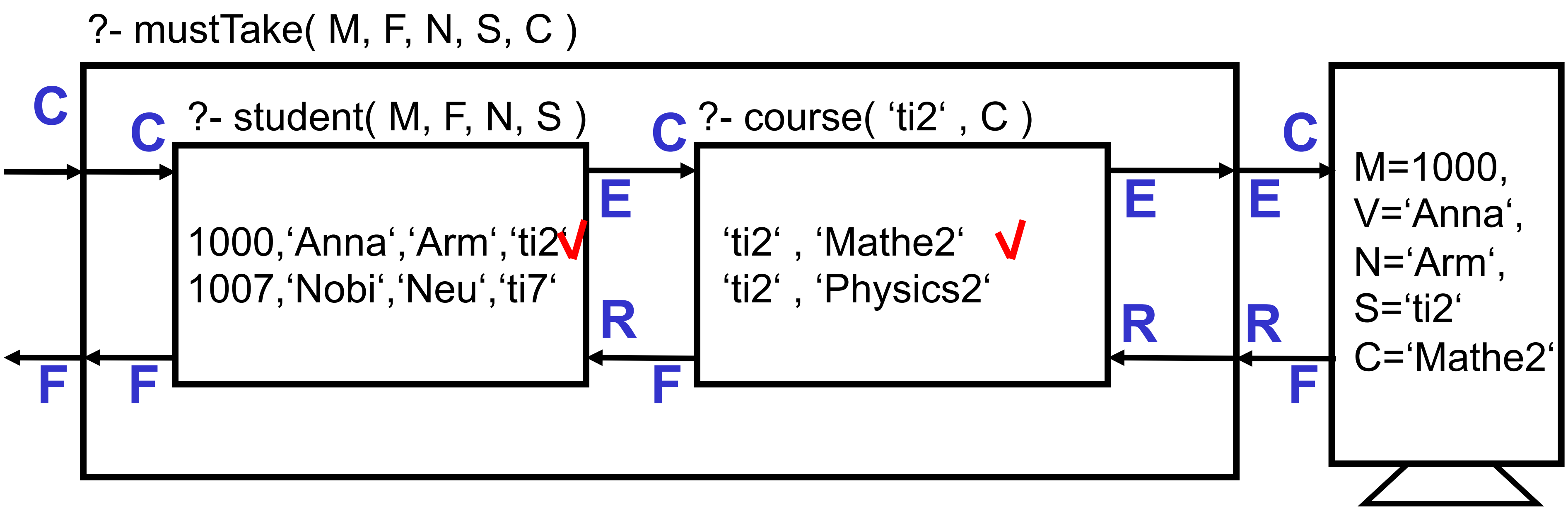
mustTake(M, F, N, S, C) :- student(M, F, N, S) , course(S, C).
?- mustTake(M, F, N, S, C).



C=Call   E=Exit   R=Redo   F=Fail

procedural semantics: 4-port model

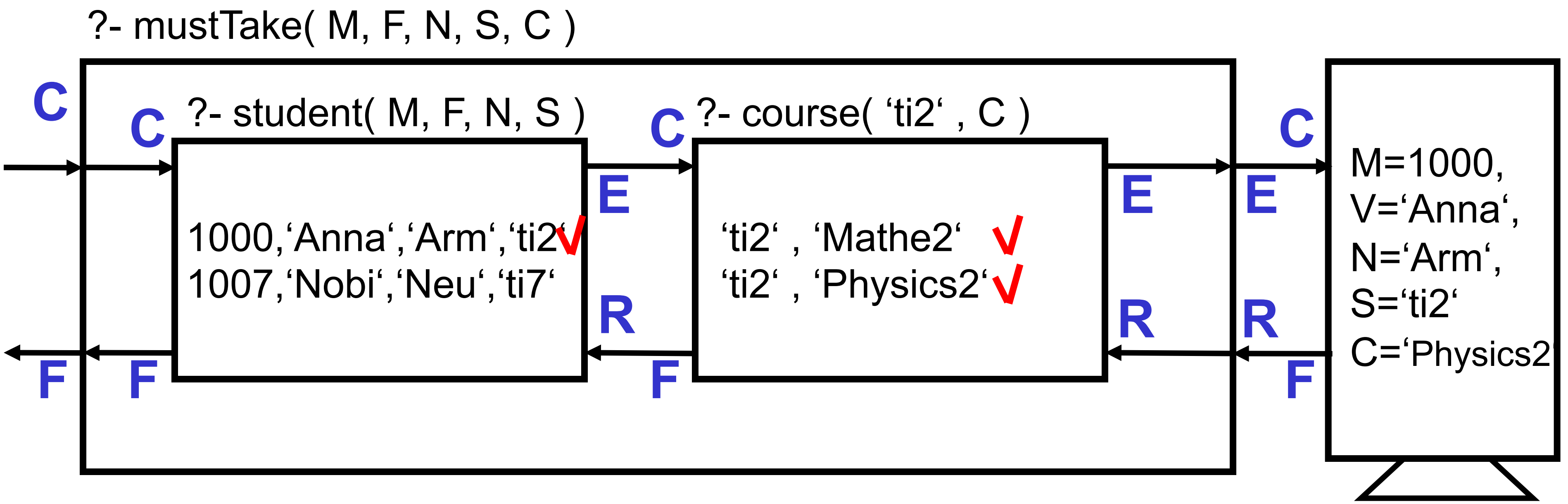
mustTake(M, F, N, S, C) :- student(M, F, N, S) , course(S, C).
?- mustTake(M, F, N, S, C).



C=Call   E=Exit   R=Redo   F=Fail

procedural semantics: 4-port model

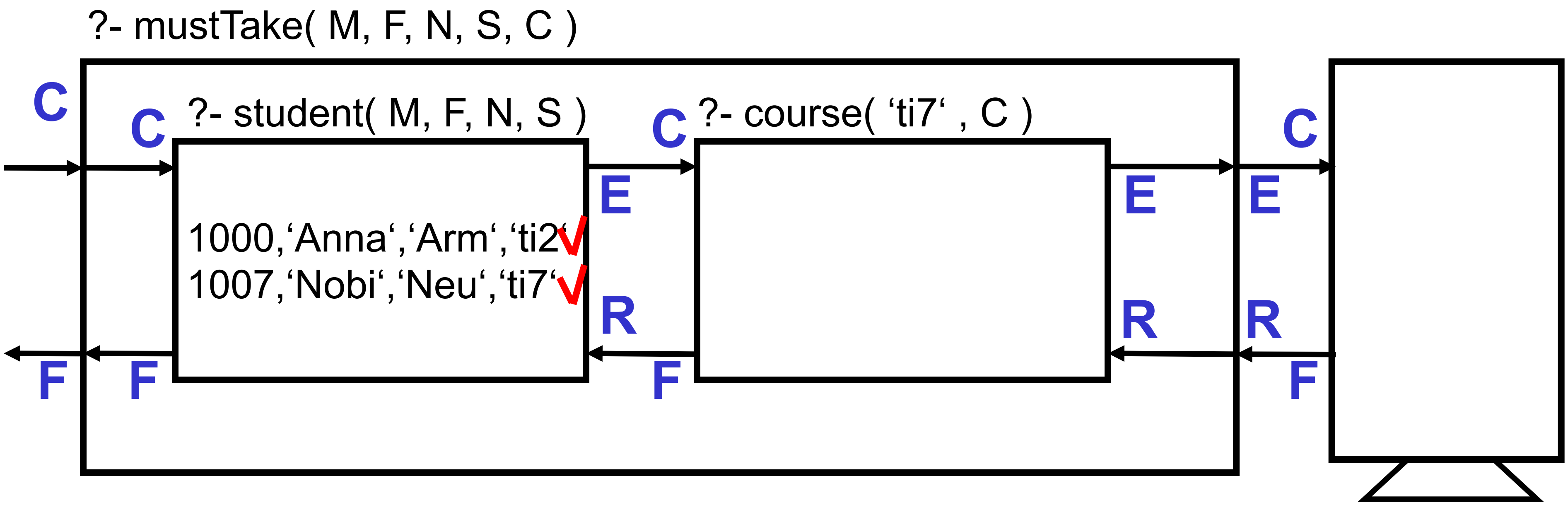
mustTake(M, F, N, S, C) :- student(M, F, N, S) , course(S, C).
?- mustTake(M, F, N, S, C).



C=Call E=Exit R=Redo F=Fail

procedural semantics: 4-port model

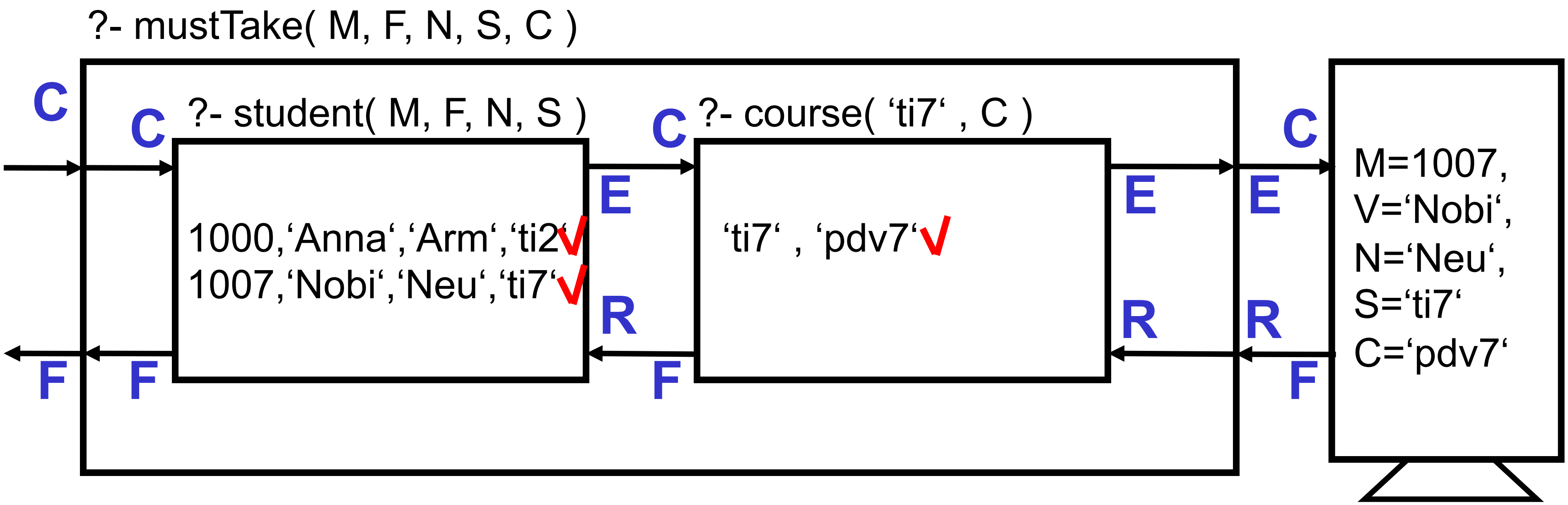
mustTake(M, F, N, S, C) :- student(M, F, N, S) , course(S, C).
?- mustTake(M, F, N, S, C).



C=Call   E=Exit   R=Redo   F=Fail

procedural semantics: 4-port model

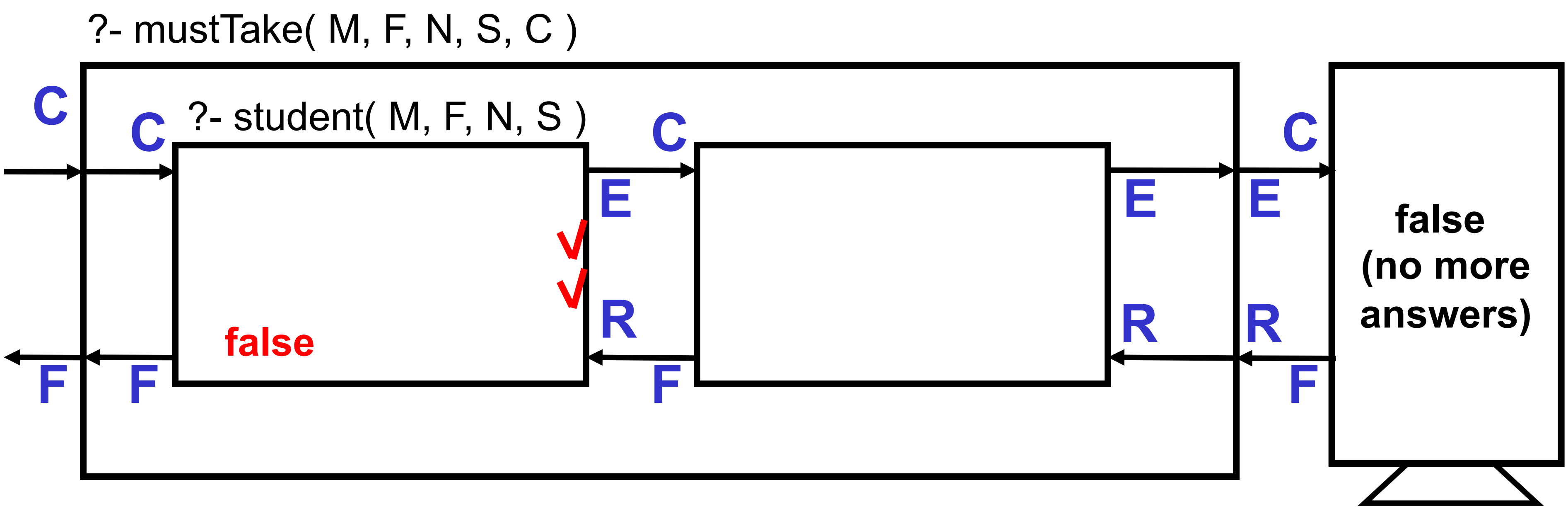
mustTake(M, F, N, S, C) :- student(M, F, N, S) , course(S, C).
?- mustTake(M, F, N, S, C).



C=Call E=Exit R=Redo F=Fail

procedural semantics: 4-port model

mustTake(M, F, N, S, C) :- student(M, F, N, S) , course(S, C).
?- mustTake(M, F, N, S, C).



C=Call E=Exit R=Redo F=Fail

Prolog‘s backtracking in Java (or C)

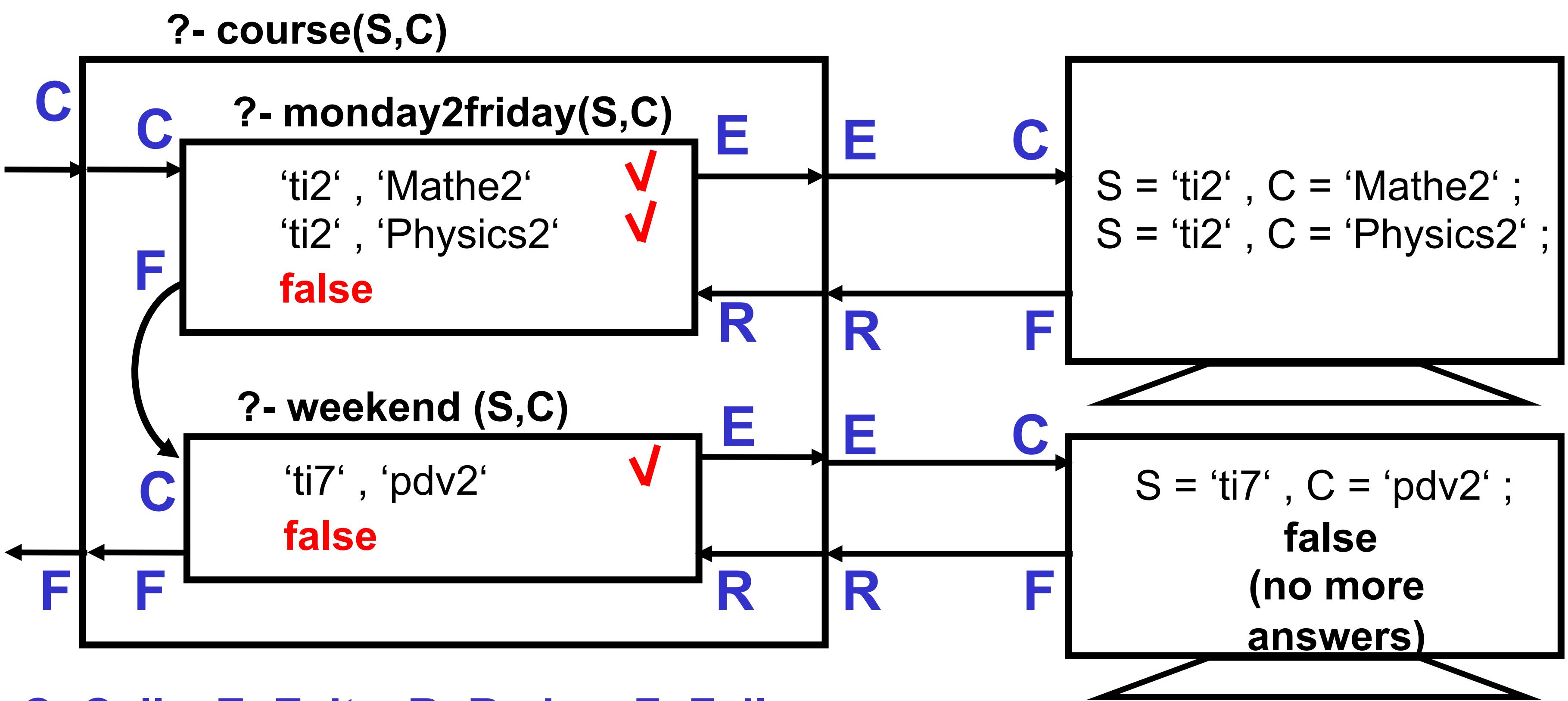
```
mustTake( M, F, N, S, C )  :-  student( M, F, N, S ) , course( S, C ) .

void mustTake( M, F, N, S, C )
{   // call-port of student
    AS =  student . getAll( M, F, N, S ) ;
    while (   student( M, F, N, S ) = AS. next( ) )
    {  // exit-port of student and call-port of course
       AK = course . getAll( S, C ) ;
       while ( course(S,C) = AK. next( ) )
       {   // exit-port of course and call-port of Output
           Output( M, F, N, S, C ) ;
           // fail-port of Output and redo-port of course
       }
       // fail-port of course and redo-port of student
    }
    // fail-port of student
}
```

4-port model with multiple clauses

given are different courses: monday2friday(S,C) and weekend(S,C)

course(S,C) :- monday2friday(S,C) .
course(S,C) :- weekend(S,C) .



C=Call E=Exit R=Redo F=Fail

Intersection, Bag Union, Difference

given are: undergraduate(S,C) **and** weekend(S,C)

intersection: weekend undergraduate courses:

```
?- undergraduate(S,C) , weekend(S,C).
```

as a rule:

```
weekendUndergraduate(S,C) :- undergraduate(S,C) , weekend(S,C).
```

Bag union: undergraduate or weekend courses with duplicates

```
undergraduateOrweekendCourse(S,C)  :-  undergraduate(S,C) .
undergraduateOrweekendCourse(S,C)  :-  weekend(S,C).
```

difference: undergraduate without weekend

```
undWithoutWe(S,C) :-  undergraduate(S,C) , \+ weekend(S,C).
```

negation operator (NOT)

Intersection, Set Union, Difference

given are: undergraduate(S,C) **and** weekend(S,C)

intersection: weekend undergraduate courses:

```
?- undergraduate(S,C) , weekend(S,C).
```

as a rule:

```
weekendUndergraduate(S,C) :- undergraduate(S,C) , weekend(S,C).
```

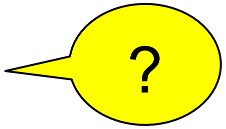
Bag union: undergraduate or weekend courses (with duplicates)

```
undergraduateOrweekendCourse(S,C)  :-  undergraduate(S,C) .
undergraduateOrweekendCourse(S,C)  :-  weekend(S,C).
```

difference: undergraduate without weekend

```
undWithoutWe(S,C) :-  undergraduate(S,C) , \+ weekend(S,C).
```

How to get the set union:
Undergrade or weekend courses without duplicates?



Negation as failure

?- \+ student(1000 , ‘Anna‘ , ‘Arm‘ , _) .
false , because
?- student(1000 , ‘Anna‘ , ‘Arm‘ , _) .
true

?- \+ student(123 , ‘Anna‘ , ‘Arm‘ , _) .
true, because
?- student(123 , ‘Anna‘ , ‘Arm‘ , _) .
false → **negation as failure**

?- \+ student(M , F , N , S) .
false, because
?- student(M , F , N , S) .
has at least one answer.

Negation as failure is different from logical negation

?- student(M , F , N , S) .
has (in general) multiple answers
returns bindings for M , F , N , and S

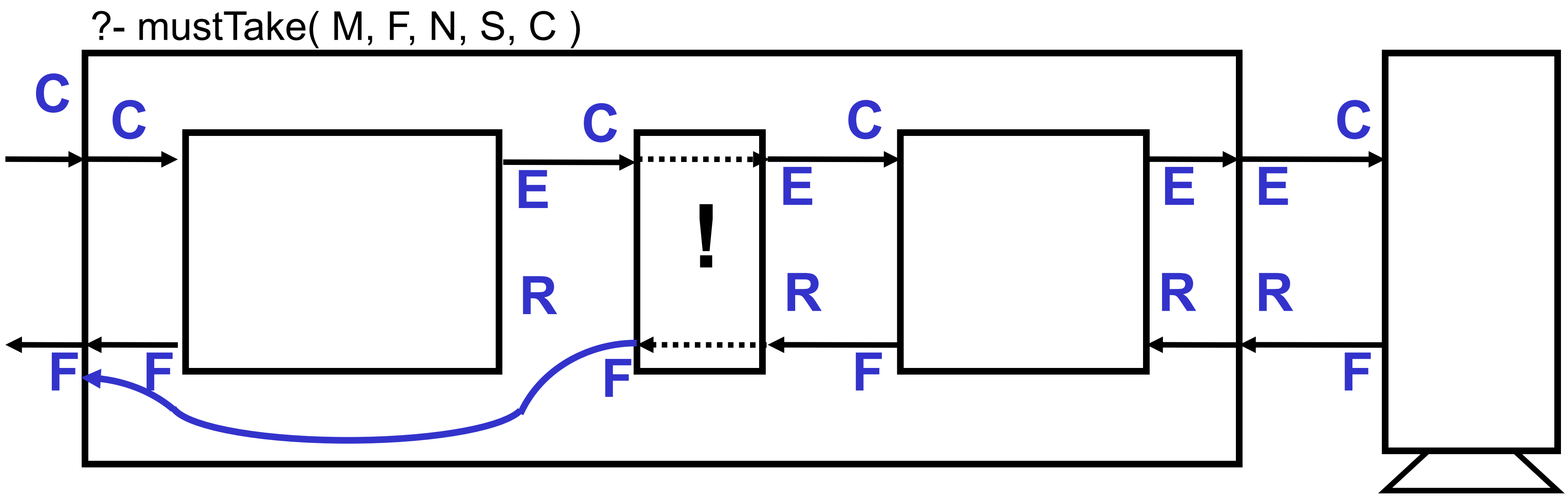
?- \+ student(M , F , N , S) .
false, because
?- student(M , F , N , S) .
has at least one answer.

?- \+ \+ student(M , F , N , S) .
true,
because
?- \+ student(M , F , N , S) .
returns false

No bindings for M , F , N , and S

Cut within the 4-port model

mustTake(M, F, N, S, C) :- student(M, F, N, S) , ! , course(S, C).
?- mustTake(M, F, N, S, C).



Cut leaves the procedure call box on the way back (=return)

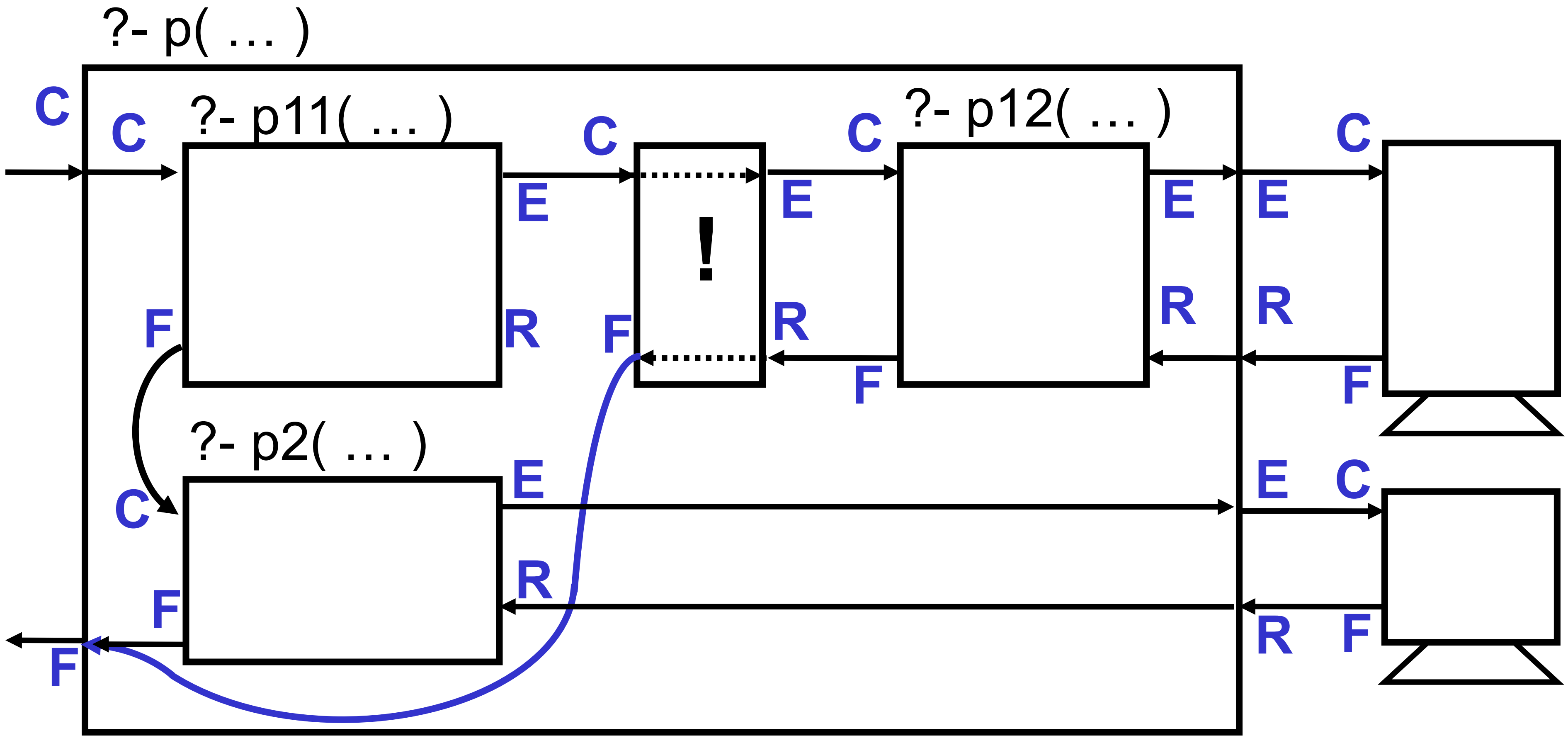
C=Call E=Exit R=Redo F=Fail

Cut in predicates with multiple rules

Cut leaves the box of the called procedure (not only the clause!)

p(…) :- p11(…) , ! , p12(…) .

p(…) :- p2(…) .



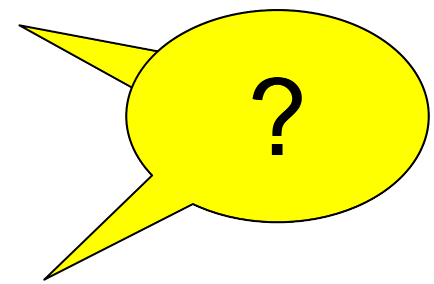
Different positions of the Cut

Find an example where it makes a difference
whether the Cut occurs early or late in a rule?

1. p(M, F, N, S, C) :- student(M, F, N, S) , course(S, C) , ! .

2. p(M, F, N, S, C) :- student(M, F, N, S) , ! , course(S, C) .

3. p(M, F, N, S, C) :- ! , student(M, F, N, S) , course(S, C) .



Find an example where it makes a difference
whether we have one or more Cuts in a rule?

4. p(M, F, N, S, C) :- student(M, F, N, S) , ! , course(S, C) , ! .

Negation as failure implemented with Cut

fail always yields false, as if implemented by

```
fail :- 2 = 3 .
```

“For semester S there is no course C offered:“

```
no_course( S , C )  :-  course( S , C ) , ! , fail .

no_course( S, C ) .
```

skip this slide now

See every solution only once

example: Which students take several courses?

Implementation of the test rule:

```
takesSeveralCourses( M ) :-
        takes( M, C1 ) , takes( M, C2 ) , \+ C1=C2 , ! .
```

0 or 1 answer per M because of Cut at the end

Implementation of the generate-and-test-rule :

```
studentTakesSeveralCourses( M, F, N, S ) :-
        student( M, F, N, S ) , takesSeveralCourses( M ).
```

generator **(generates every student exactly once)** — **test** **(selects or does not select)**

Query:

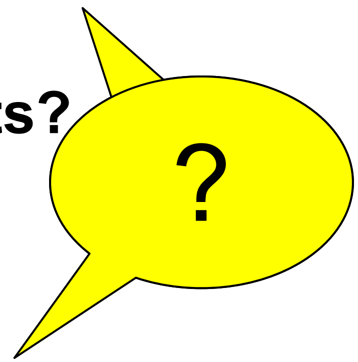
```
?- studentTakesSeveralCourses( M, F, N, S ) .
```

Exercises

Assume, we have a relations takes(M, C) and course(S, C)
M is Matriculation number, C is Course, S is Semester

Assume further, C is a key of the relation course,
use the generate and test approach in the following queries:

1. **Which courses are taken by more than one students?**
2. **Which courses are taken by less than two students?**
3. **Which courses are taken by exactly one student?**
4. **Which courses are taken by exactly two students?**



Replace ∀x∈R(p(x)) with not ∃x∈R(not p(x))

example: Which students take *all* courses offered for 'ti2' ?

{ (M,F,N,S) ∈ Student | ∀('ti2',C) ∈ course (takes(M,C)) } ⇔
{ (M,F,N,S) ∈ Student | not ∃('ti2',C) ∈ course (not takes(M,C)) }

generate-and-test-rule :

```
studentTakesAllCoursesOfferedForti2( M, F, N, S ) :-
    student( M, F, N, S ) , \+ atLeastOneti2CourseNotTakenBy( M ) .
```

generator (generates every student exactly once)

test (selects or does not select)

Test rule implementation :

```
atLeastOneti2CourseNotTakenBy( M ) :-
    course( 'ti2', C ) , \+ takes( M, C ) , ! .
```

Query:

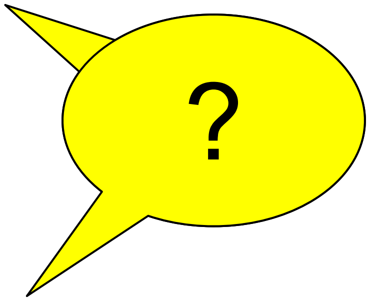
```
?- studentTakesAllCoursesOfferedForti2( M, F, N, S ) .
```

Exercises

Assume, we have a relations takes(M, C) , course(S, C) , and student(M , F , N , S)
M is Matriculation number, C is Course, S is Semester,
F is the first name, N is the last name of a student

Assume further, C is a key of course, M is key of student.
Use the generate and test approach in the following queries:

1. **Which students take all courses ?**
   Use your slides printout to ‘copy‘ this solution



2. **Which courses are taken by all students ?**

3. **Which courses are taken by all students having first name ‘Peter‘?**

Replace maximum with "≥ all"

example: Which student has the highest student ID ?

$\{ (M,F,N,S) \in Student \mid M = max(\{ M2 \mid (M2,V2,N2,S2) \in Student \}) \}$ $\Leftrightarrow$

$\{ (M,F,N,S) \in Student \mid \forall (M2,V2,N2,S2) \in Student (M \geq M2) \}$ $\Leftrightarrow$

$\{ (M,F,N,S) \in Student \mid not\ \exists (M2,V2,N2,S2) \in Student (M < M2) \}$

Generate-and-test-rule :

```
studentHasHighestMnr( M , F , N , S ) :-
        student( M, F, N, S ) , \+ someoneHasHigherMnrThan( M ) .
```

generator (generate every student exactly once)

test (selects or does not select)

Test rule implementation :

```
someoneHasHigherMnrThan ( M ) :- student( M2 , _ , _ , _ ) , M < M2 .
```

Query:

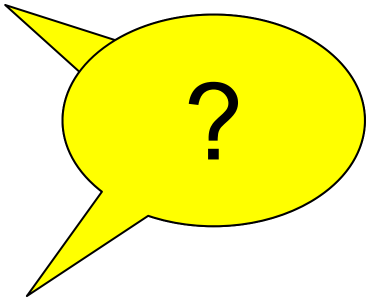
```
?- studentHasHighestMnr( M , F , N , S ) .
```

Exercises

Assume, we have a relations takes(M, C) , course(S, C) , and student(M , F , N , S)
M is Matriculation number, C is Course, S is Semester,
F is the first name, N is the last name of a student

Assume further, C is a key of course, M is key of student.
Use the generate and test approach in the following queries:

1. **Which students in semester ‘ti2‘ have the highest matriculation number?**
   Use your slides printout to ‘copy‘ this solution



2. **Which of students taking the course ‘Physics2‘ have the highest matriculation number?**

Practical work with the SWI-Prolog system

1st window (SWI-Prolog) for queries and calling the editor!

```
MyPrologDirectory > swipl -s exercise1.pl
Welcome to SWI-Prolog (threaded, 64 bits, version 8.0.2) ...

?- edit.
true.

?- orders(smith,Order).
Order = item1 ;
Order = item2

?- halt.
MyPrologDirectory >
```

2nd window SWI-Prolog editor: for database facts and rules, i.e. your program, and for calling the compiler

exercise1.pl [modified]

File Edit Browse Compile Prolog Pce Help

exercise1.pl [modified]

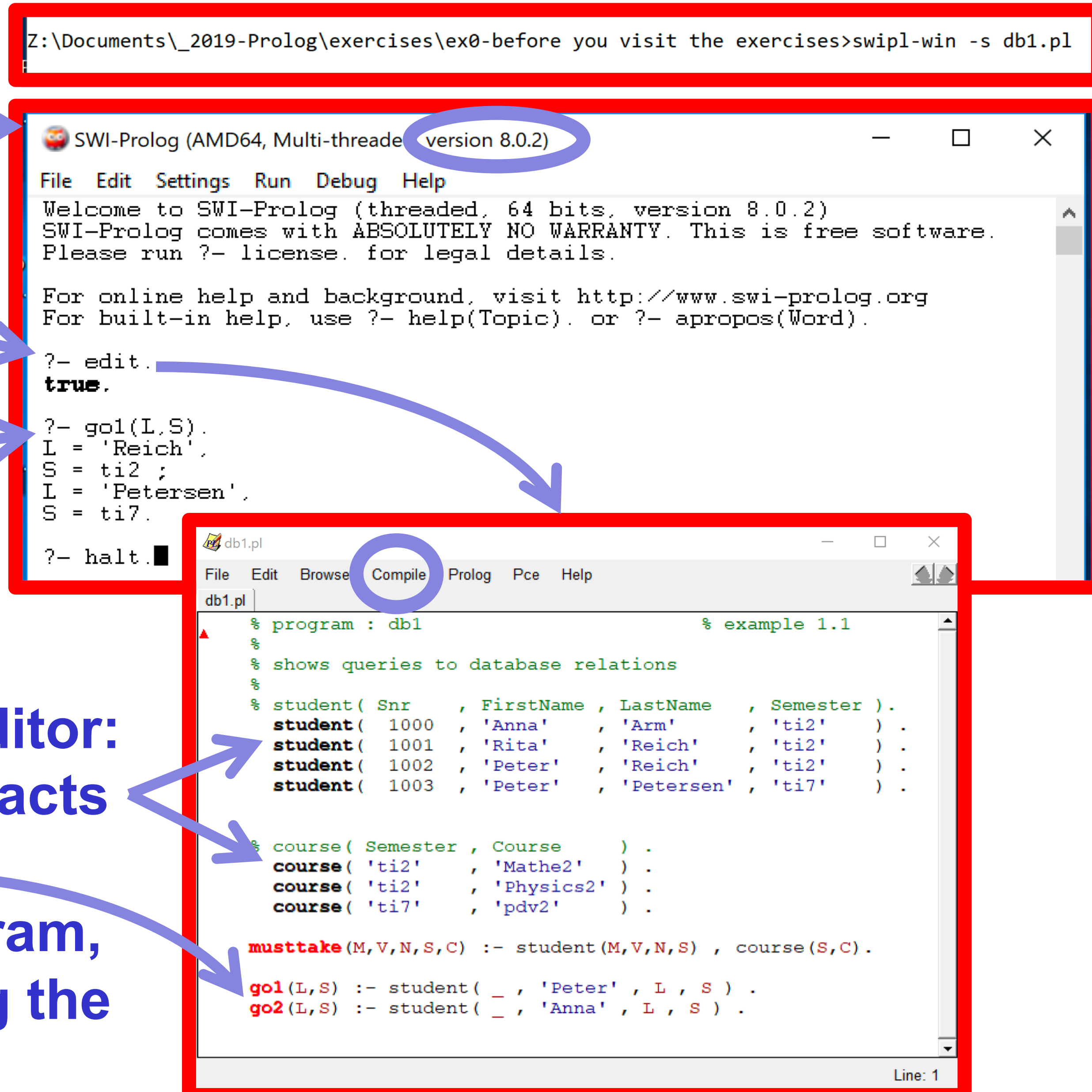
```
% supplies( Supplier  , Item  , Days , Price ).
supplies(   supplier1 , item1 ,   3  , 500   ).
supplies(   supplier1 , item2 ,   6  , 200   ).
supplies(   supplier1 , item2 ,   3  , 300   ).
supplies(   supplier2 , item2 ,   4  , 500   ).

% orders( Customer , Item  ).
orders(   smith    , item1 ).
orders(   smith    , item2 ).
orders(   meier    , item3 ).

% The desired solution is e.g.:
q1( Item ) :-supplies(  , Item ,   , P ) , P =< 220 .
```

q1/1: (not loaded) unreferenced   Line: 13

Practical work with SWI-Prolog using Windows



1st window (SWI-Prolog) for calling the editor! and for queries

2nd window SWI-Prolog editor: for database facts and rules, i.e. your program, and for calling the compiler

Practical work with SWI-Prolog using Windows

```
Z:\Documents\_2019-Prolog\exercises\ex0-before you visit the exercises>swipl-win -s db1.pl
```

1st window (SWI-Prolog) for calling the editor! and for queries

```
SWI-Prolog (AMD64, Multi-threaded, version 8.0.2)
File  Edit  Settings  Run  Debug  Help
Welcome to SWI-Prolog (threaded, 64 bits, version 8.0.2)
SWI-Prolog comes with ABSOLUTELY NO WARRANTY. This is free software.
Please run ?- license. for legal details.

For online help and background, visit http://www.swi-prolog.org
For built-in help, use ?- help(Topic). or ?- apropos(Word).

?- edit.
true.

?- go1(L,S).
L = 'Reich',
S = ti2 ;
L = 'Petersen',
S = ti7

?- halt
```

2nd window SWI-Prolog editor: for database facts and rules, i.e. your program, and for calling the compiler

```
db1.pl
File  Edit  Browse  Compile  Prolog  Pce  Help
db1.pl
% program : db1                              % example 1.1
%
% shows queries to database relations
%
% student( Snr    , FirstName , LastName   , Semester ).
  student(  1000  , 'Anna'    , 'Arm'      , 'ti2'     ) .
  student(  1001  , 'Rita'    , 'Reich'    , 'ti2'     ) .
  student(  1002  , 'Peter'   , 'Reich'    , 'ti2'     ) .
  student(  1003  , 'Peter'   , 'Petersen' , 'ti7'     ) .


% course( Semester , Course      ) .
  course( 'ti2'    , 'Mathe2'    ) .
  course( 'ti2'    , 'Physics2'  ) .
  course( 'ti7'    , 'pdv2'      ) .

musttake(M,V,N,S,C) :- student(M,V,N,S) , course(S,C).

go1(L,S) :- student( _ , 'Peter' , L , S ) .
go2(L,S) :- student( _ , 'Anna' , L , S ) .

Line: 1
```

Summary

Prolog supports different programming styles:

1. **Procedural style (using Cut(!) and Negation as Failure (\+))**
   This allows for queries containing
   all, at most one, min, max, exactly one, … .
   And this allows to avoid duplicate answers, if we have a generator relation for the superset in which we search, i.e. agenerator that generates each candidate exactly once

   (You will need the procedural style for Exercise 1.)

2. **Declarative style (NOT using Cut or Negation as Failure)**
   This allows for cleaner (pure!) Prolog programming

   (You will need the declarative style for Exercise 2.)