Rave I

Developed by
Jeppesen Crew Academy

for version 22 of Crew Pairing, Crew Rostering
and Tail Assignment

Practical Details

Restrooms

Breaks

Phones

Wifi

Lunch arrangements

Quiz

Evaluation

Participants presentation

- Name, company
- Role
- Experience
- Your expectations
- <Other>

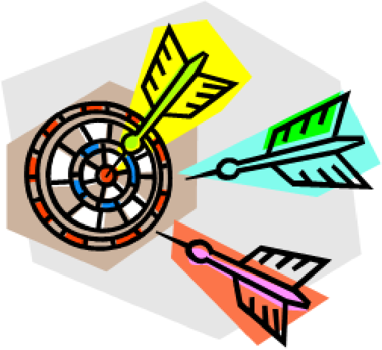
Course goals

This course will teach you:

- Rave syntax
- How to write Rave code and how it is used
- About variables, parameters, rules and costs
- How to find and use the on-line help and the Rave reference manual

You will know how to:

- Update and maintain existing code
- Implement new functionality with Rave



Prerequisites

- Any Product I course
- Min 12 months of real life programing experience
- Knowledge about the airline/rail business side.

Please –

Don't be afraid to ask questions if anything is unclear!

Course material

Course slides

Course manual

Online documentation

Code standards

Agenda Day 1

09:00 - 10:15	Introduction to Rave Rave programmers toolkit
10:15 - 10:30	**Coffee break**
10:30 - 12:30	Data types, keywords Variables, Parameters
12:30 - 13:30	**Lunch**
13:00 - 15:00	Functions Built-in functions
15:00 - 15:15	**Coffee break**
15:15 - 17:00	If-Then-Else, Tables Etables and sets

All times are approximate – changes may/will occur

Short breaks every ~40 minutes or so

Agenda Day 2

09:00 - 10:15	Review of day 1 Levels Traversers
10:15 - 10:30	**Coffee break**
10:30 - 12:30	Void values, Filters Modules and DWS
12:30 - 13:30	**Lunch**
13:30 - 15:00	Rules
15:00 - 15:15	**Coffee break**
15:15 - 17:00	more Rules

All times are approximate – changes may/will occur

Short breaks every ~40 minutes or so

Agenda Day 3

09:00 - 10:15	Review of day 2 Costs
10:15 - 10:30	**Coffee break**
10:30 - 12:30	more Costs
12:30 - 13:30	**Lunch**
13:30 - 15:00	Contexts, Iterators
15:00 - 15:15	**Coffee break**
15:15 - 17:00	Map variables Summary Evaluation

All times are approximate – changes may/will occur

Short breaks every ~40 minutes or so

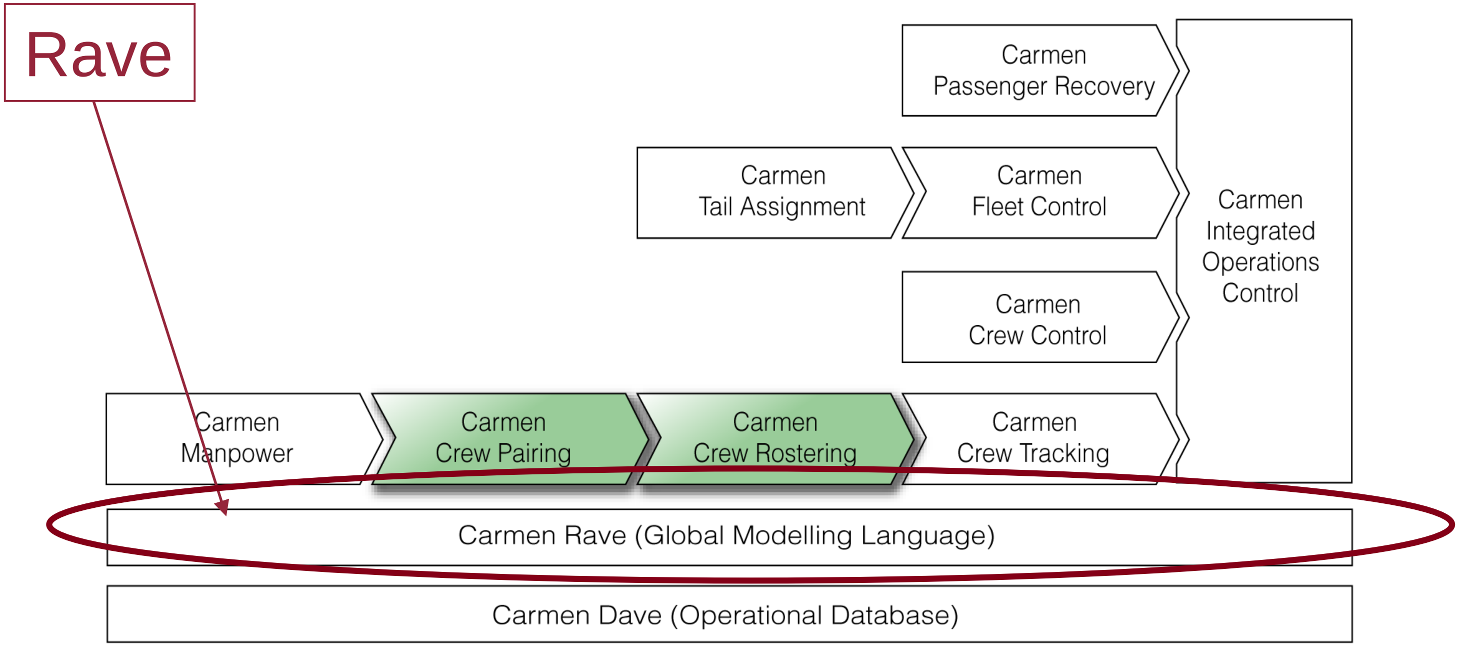
Chapter 1

Jeppesen Products

What is Rave

Rave Toolkit

Jeppesen Products



Rave

Carmen Passenger Recovery

Carmen Tail Assignment

Carmen Fleet Control

Carmen Crew Control

Carmen Integrated Operations Control

Carmen Manpower

Carmen Crew Pairing

Carmen Crew Rostering

Carmen Crew Tracking

Carmen Rave (Global Modelling Language)

Carmen Dave (Operational Database)

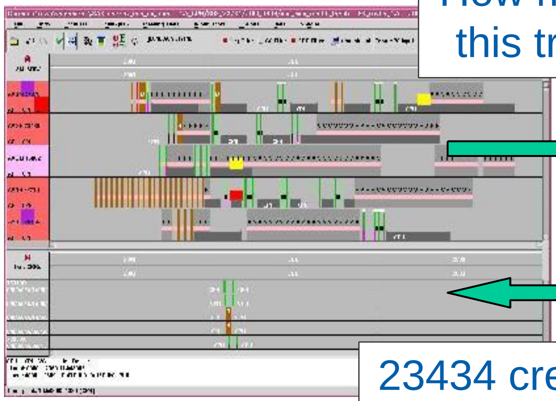
Jeppesen Products

Studio ↔ Rave ↔ Optimizer

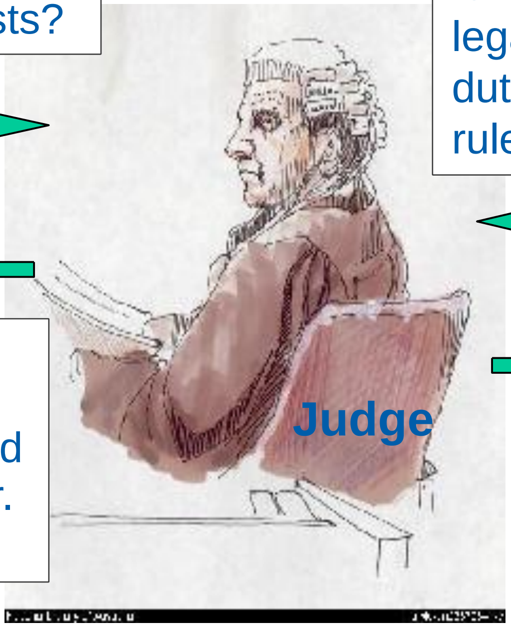
Studio

Rave

Optimizer



How much does this trip costs?

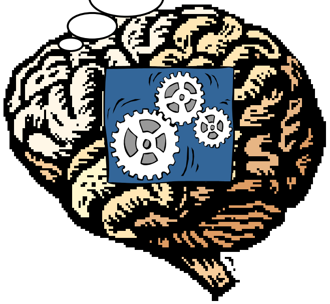


Is this roster legal according to duty_max_active_flights rule?

Minimize $\boldsymbol{c}^T\boldsymbol{x}$
Subject to: $A\boldsymbol{x} = \mathbf{1}$
$\mathbf{x_k} \in [\mathbf{0,1}]$

23434 crew
+ 3432 hotel
+ 300 deadhead
+ 430 ground tr.
= 27596

Yes



What is Rave?

Computer language

- developed and maintained by Jeppesen Systems.

Special notation and syntax

Used to model a planning problem

- legality
- objectives
- presentations.

Rave

Rave is:

- Functional language (Haskell, ML) without recursion
  - However, extensive use of function calls should be avoided
- No side effects
  - does not ‘do’ anything
  - only answers questions
- Domain specific, developed for crew planning
- Limited, not Turing complete (for simplicity and security).

What is Rave?

Rave in Studio

In Studio, Rave is used for:

- searching
- drawing of activities
- displaying information
- providing data for reports (Rave Publisher)
- legality control when building solutions manually.

Note: Studio provides the data, Rave just evaluates.

Rave in optimization

Limits automatic construction:

- horizontal constraints (rules)
- vertical constraints (global conditions).

Guides optimization to good quality solutions:

- cost function
- sorting objects.

What is Rave?

Rave example

There is an union agreement that limits the maximum number of flights pilots may fly in a day.

Regulation:

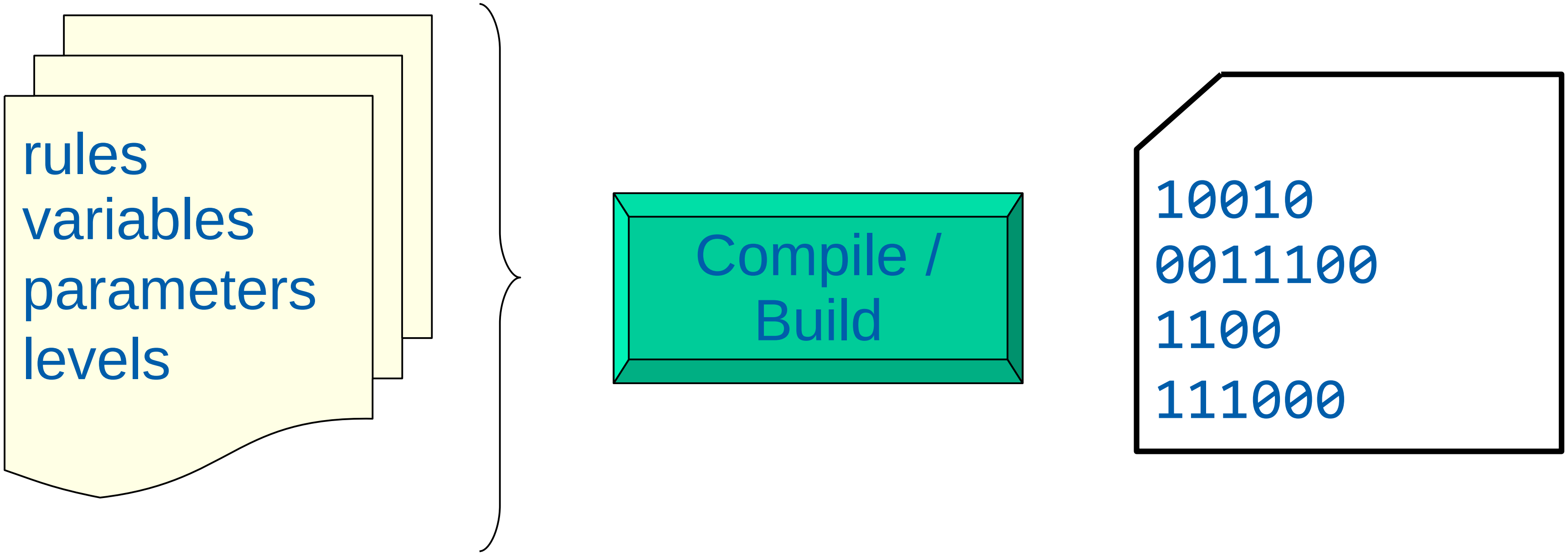
”In a day, crew can have no more than 4 active flights.”

What is Rave?

Rave example

```
rule duty_max_active_flights_rule =
   %duty_active_flights%
   <= %duty_max_active_flights_p%;
   remark ”L09: Limit active flights per day: ”;
end


%duty_max_active_flights_p% = parameter 4
   remark ”L09_p1: Max number of“
          “ active flights per day: ”;
%duty_active_flights% =
   count(leg(duty)) where (not deadhead);
```

Rave


rules
variables
parameters
levels
Compile /
Build
10010
0011100
1100
111000

User Interaction with Rave

Rave programmer:

- creates new rules/parameters.

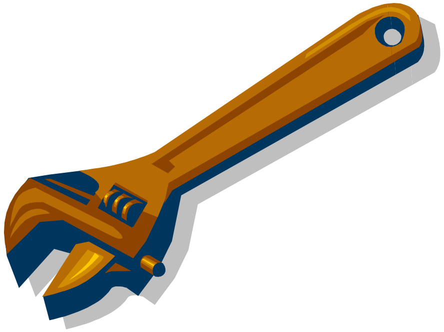
Planner:

- toggles rules on/off
- changes parameter values.

Rave Toolkit

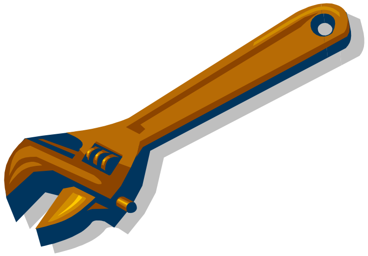
Let’s look at the Studio Rave toolkit:

- Compile rules
- Rave documentation
- Load, Unload and work with Rules
- Show Rule Values
- Rave Explorer
- Rave Evaluator



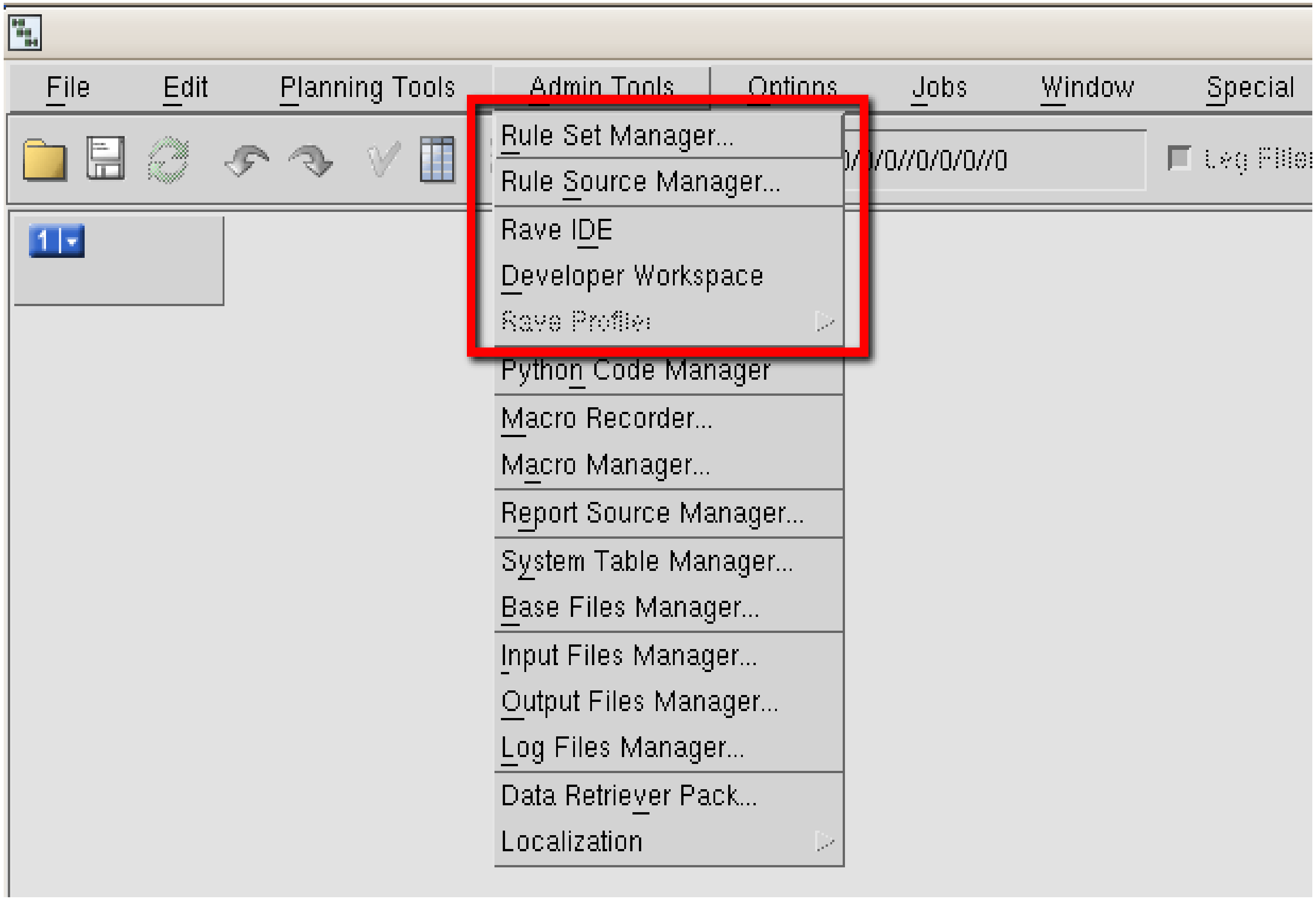
Demonstration

Rave Toolkit



Rave Toolkit Demo

Admin Tools menu



Rule Set Manager

Rule Set Manager contains all compiled rule sets

You can:

- load a rule set
- view info about the last build

Rule Source Manager

Rule Source Manager contains all source files

You can:

- view / edit production Rave source files
- compile rule sets

Rave Toolkit Demo

Rule Source Manager

Compiler (syntax and type checking):

- **Build** (build for Studio and optimization)
- **Build…** (selective build).

Documentation

Documentation:

- Rave Reference Manual
- **Help > Keywords etc.** provides help about keywords, contexts, transforms and iterators.

Development environments:

- DWS (Developer Workspace)
- Rave IDE (Integrated Development Environment)
- Rave mode in Emacs (called CRC mode).

Rave Toolkit Demo

Loading, reloading rule sets

Loading a rule set resets the parameters to their default values (as defined in the rave code)

Reloading a rule set keeps the current parameter settings. Use reload during development.

Load a rule set:

Rule Set Manager

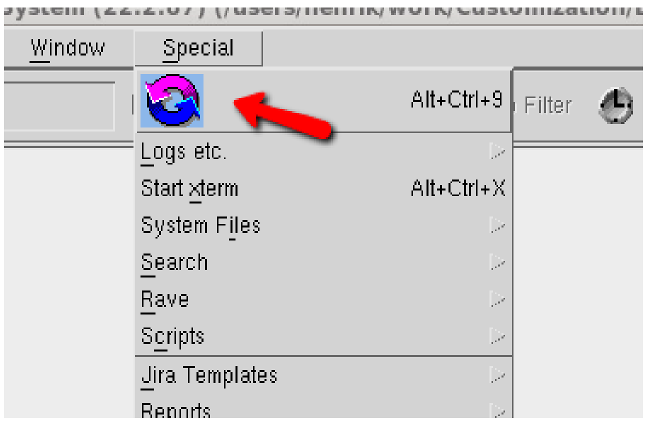
Or

File > Load > Load Rule Set > [Name]

Reload current rule set:

File > Load > Reload Rule set

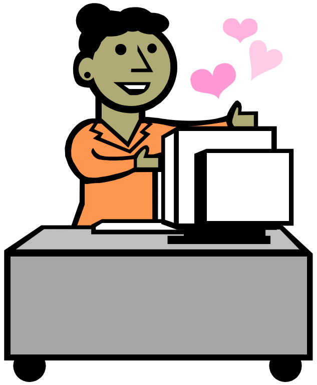
or



Exercise 1

~45 mins

Exercise 1 summary



Chapter 2

Data Types

Keywords

Comments

Reserved Words

Data types

`Int` **whole numbers:**

```
7, 234, -300
<2 000 000 000 (-2^32..0..2^31)
```

`Bool` **truth values:**

```
True, False
```

`String` **sequence of characters:**

```
”A string”, ”%The_2nd_string%”, ”Y”
```

`Enum` **enumeration type**
defines a new set of values:

```
enum detail_level =
     high;
     medium;
     low;
end
```

Data Types

Dates

`Abstime` **date and time of an event:**

```
23Jun1998 16:45
10jan2003 (0:00)
```

`Reltime` **time of day, a period of time between two events or the duration of an event:**

```
1:45, 72:00, -0:05
```

Data Types

Dates in the Jeppesen system

First possible date is 01JAN1901*

Last possible date is 31DEC2099*

First minute of a day is 0:00

Last minute of a day is 23:59

*Older versions of Rave and components that Rave interacts with may have different limitations!

Keywords

- used for accessing object information e.g: `arrival`, `flight_number` and `user`
- should be seen as basic attributes on objects
- data you just have to know about
- defined by the applications
- all the data provided by the applications
- used to create more advanced expressions.

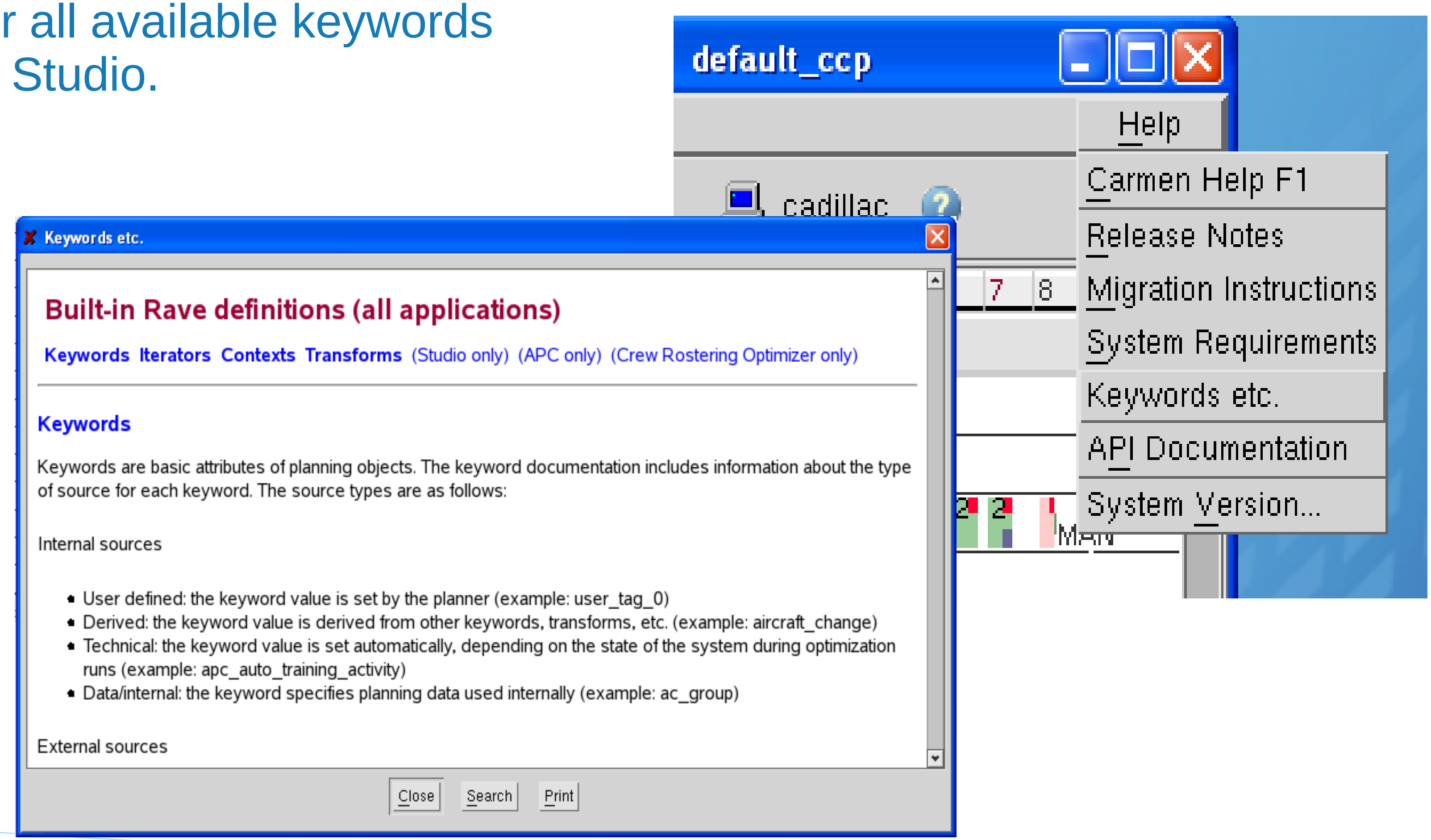
The keyword `arrival` will give the arrival time of the current leg in UTC: `arrival = 21jan1998 08:45`

More keywords

```
aircraft_type
number_of_business_class_seats
deadhead
local_arrival_time_summer
departure_airport_name
ground_duty
```

Keywords
Documentation

There is online help
for all available keywords
in Studio.



Built-in Rave definitions (all applications)

Keywords Iterators Contexts Transforms (Studio only) (APC only) (Crew Rostering Optimizer only)

Keywords

Keywords are basic attributes of planning objects. The keyword documentation includes information about the type of source for each keyword. The source types are as follows:

Internal sources

- User defined: the keyword value is set by the planner (example: user_tag_0)
- Derived: the keyword value is derived from other keywords, transforms, etc. (example: aircraft_change)
- Technical: the keyword value is set automatically, depending on the state of the system during optimization runs (example: apc_auto_training_activity)
- Data/internal: the keyword specifies planning data used internally (example: ac_group)

External sources

Keywords

Data types

Each keyword will automatically have a data type:

– `flight_number`	**`Int`**
– `deadhead`	**`Bool`**
– `departure`	**`Abstime`**

Mixing data types

- Some data types cannot be mixed

  `234 + True` would not mean anything

- Times and integers may be mixed when it makes sense:

```
3jan97 4:00 + 5:45 = 3jan97 9:45
0:05/5 = 0:01
0:05/0:01 = 5
120 * 0:01 = 2:00
24:00 * 7 = 168:00
24:00 / 0:01 = 1440
```

Data Types
Real Numbers

Rave does not handle real numbers (such as 3.14), all decimals are truncated.

```
3.14 → 3
10.999 → 10
9.9 + 9.9 →  18
```

Workaround:
Work with a multiple of 10 (or more) of the actual value
Divide with the multiple at the end

Int value: `(99 + 99)/10 = (198)/10 = 19`

Decimal: `(99 + 99) mod 10 = 8`

Round a sum of floating values to the closest integer:
add magnified terms, add 5, divide by 10

```
((99 + 99) + 5 )/10 = (203)/10 = 20
```

Comments

Use comments in the Rave code to increase readability:

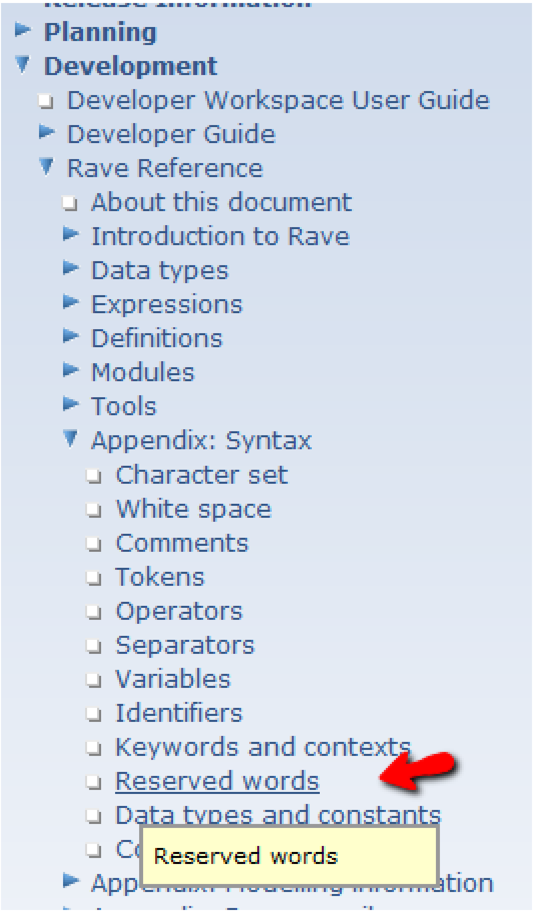
```
/* This is a comment */

/*
 * Use comments to increase readability
 * in your code and make it easier
 * to maintain /HM
 */

/* End of file */
```

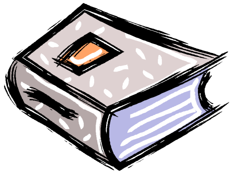
Reserved words

Some words are reserved for the Rave language.

Examples:
`abs, use, export, end`

These words have a special meaning and may not be redefined or used in a different way than intended.



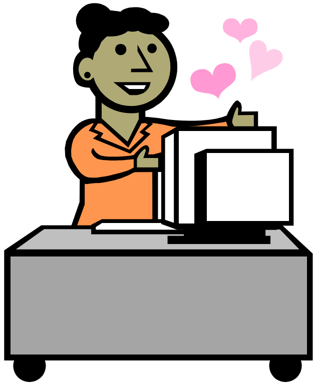


See
Help> Development> Rave Manual > Appendix: Syntax
for more information about Reserved Words

Exercise 2

~15 mins

Exercise 2 summary



Chapter 3

Variables

Parameters

Variables

- Rave is a language that defines *variables* (attributes) for objects
- Each variable is an expression, which can use the object’s other variables to calculate its own value
- The code is order independent
- Every variable has a value for every object
- A variable has the same value for an object during the whole calculation – remember, rave does not *do* anything.

Variables

Variables are the names assigned to value- and function definitions. They can be…

- constants
- parameters
- calculations using other variables
- functions.

Variables
Constants

Constants are used when a value never changes:

```
%min_time_btw_duties% = 8:00;
```

In this case, the programmer does not have to know that the value is 8 hours. In other places of the code he may simply use the constant.

Parameters

Parameters give a planner possibility to instantly interact with Rave without recompiling the rule set:

```
%min_time_btw_duties% = parameter 8:00;
```

Parameters

Parameter attributes guide the Planner to setting good values:

```
%min_time_btw_duties% = parameter 8:00
    minvalue 8:00
    maxvalue 20:00
    remark “Min time between duties”;
```

A Planner may not change the value to 8, he may not change the data type!

Variables

Calculations

This is a most common form of a variable:

```
%leg_time% = %leg_end% - %leg_start%;
%leg_end% = …
```

It is used when the value depends on other values.

Names

Case insensitive

Use mainly lowercase characters: `a, b, c... z`

underscore: `_`

and numbers `0, 1, 2,` …9

Choose a name indicating what the value is for

```
underscores_btw_words
```

Variables

Names

```
%this_is_a_good_name% = …
%extra_work_time% =
   %briefing% + %debriefing%;


%Bad% = …
%x% = %b% + %d%;
```

Variables

Syntax

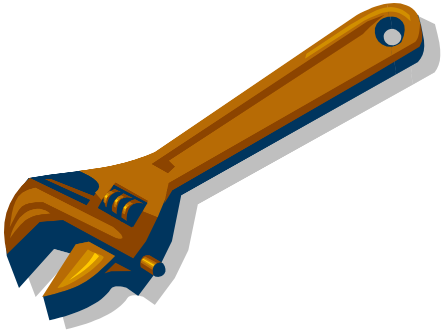
```
%identifier%[(DataType1 arg1,…)] =
    [let local1 = expr1,…;]
    [parameter]
    [minvalue const1] [maxvalue const2]
    definition
    [remark ”Text”
    [, planner ”Text”];]
```

Variables

Parameters

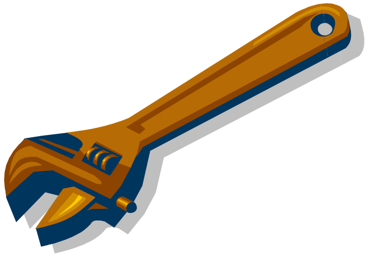
Let's look at the Studio Parameter Form:

- Simple parameters
- Parameters with/without remarks
- Parameters with bounds



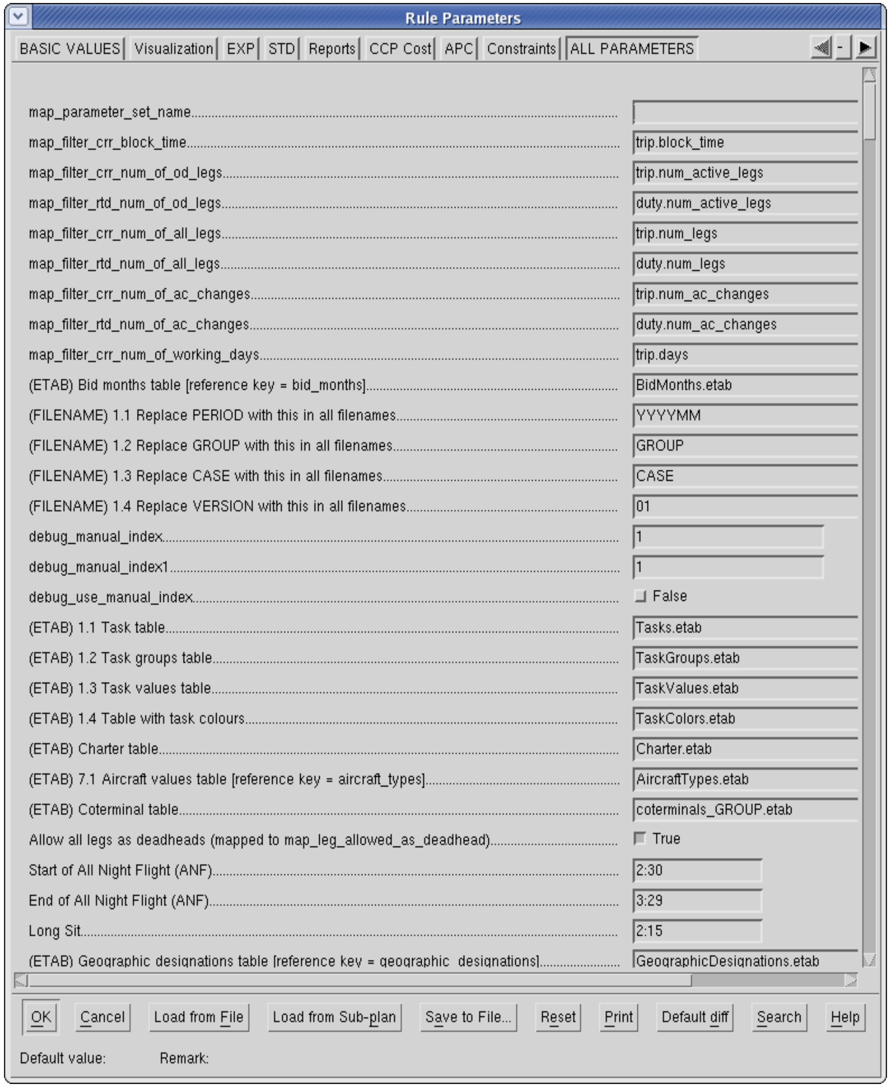
Demonstration

Studio Parameter Form



Studio Parameter Form Demo

Parameter Form



map_parameter_set_name	
map_filter_crr_block_time	trip.block_time
map_filter_crr_num_of_od_legs	trip.num_active_legs
map_filter_rtd_num_of_od_legs	duty.num_active_legs
map_filter_crr_num_of_all_legs	trip.num_legs
map_filter_rtd_num_of_all_legs	duty.num_legs
map_filter_crr_num_of_ac_changes	trip.num_ac_changes
map_filter_rtd_num_of_ac_changes	duty.num_ac_changes
map_filter_crr_num_of_working_days	trip.days
(ETAB) Bid months table [reference key = bid_months]	BidMonths.etab
(FILENAME) 1.1 Replace PERIOD with this in all filenames	YYYYMM
(FILENAME) 1.2 Replace GROUP with this in all filenames	GROUP
(FILENAME) 1.3 Replace CASE with this in all filenames	CASE
(FILENAME) 1.4 Replace VERSION with this in all filenames	01
debug_manual_index	1
debug_manual_index1	1
debug_use_manual_index	☐ False
(ETAB) 1.1 Task table	Tasks.etab
(ETAB) 1.2 Task groups table	TaskGroups.etab
(ETAB) 1.3 Task values table	TaskValues.etab
(ETAB) 1.4 Table with task colours	TaskColors.etab
(ETAB) Charter table	Charter.etab
(ETAB) 7.1 Aircraft values table [reference key = aircraft_types]	AircraftTypes.etab
(ETAB) Coterminal table	coterminals_GROUP.etab
Allow all legs as deadheads (mapped to map_leg_allowed_as_deadhead)	☑ True
Start of All Night Flight (ANF)	2:30
End of All Night Flight (ANF)	3:29
Long Sit	2:15
(ETAB) Geographic designations table [reference key = geographic_designations]	GeographicDesignations.etab

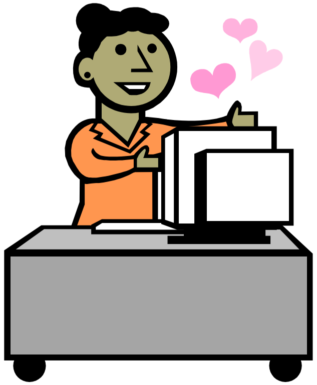
Default value:

Remark:

Exercise 3

~60 mins

Exercise 3 summary



Chapter 4

Functions

Built-in functions

Functions

Functions make variables more flexible

```
%time_at_night%(reltime a_time) =
    a_time > 22:00
    or
    a_time < 06:00;
```

Functions

Examples

```
%even_number%(int i) = i mod 2 = 0;
```

Usage example:

```
%trip_has_even_number_of_days% =
    %even_number%(%trip_days%);
```

```
%circle_area%(int r) =
    let pi = 314;
    r*r*pi / 100;
```

Let variables are temporary help variables that keep their initial value throughout the evaluation. Once initialized, they cannot be changed!

Built-in functions

There are a number of predefined functions that provide special services:

- numerical
- string and formatting
- date/time functions.

Numerical

Numerical Built-in Functions:

If you need to know the smallest of the variables %a% and %b% you may use nmin():

```
    %smallest_a_b% = nmin(%a%, %b%);
```

The absolute value of %a%:

```
    %positive_a% = abs(%a%);
```

Built-in Functions

String and Formatting

String Concatenation:

```
    concat(s1, s2, ..., sn)
```

returns a merged string of two or more smaller strings

```
concat("Carmen"," ","Systems")
  ->"Carmen Systems"
```

Formatting Functions:

```
    format_int(int, format string)
```

formats an integer as a string in a flexible way

```
format_int(123, "x=%4d") -> "x= 123"
format_int(123, "x=%-4d") -> "x=123 "
```

Built-in Functions

Time

```
round_[up,down]_[week,month,year](abstime)
round_[up,down](value, step)

time_of_[day,week](abstime)
add_[weeks,months,years](abstime, int)

overlap, scale_time
```

See
Rave Reference > Expressions > Built-in functions
for more information

Built-in functions

Date/Time Functions

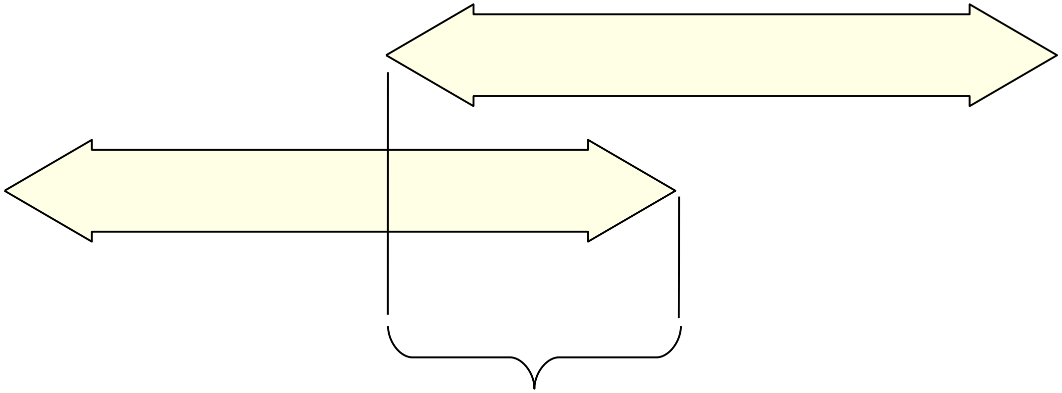
```
round_down(value, step)
round_up(value, step)
```

Return the given value (integer, reltime or abstime) rounded up/down the nearest multiple of the given step (integer or reltime)

```
round_down(15Nov2008 10:00, 24:00)
  -> 15Nov2008 0:00

round_up(15Nov2008 10:00, 24:00)
  -> 16Nov2008 0:00
```

Built-in functions
Overlap

Calculate how many minutes two time periods overlap…



Overlap

Built-in functions

Overlap

…calculate the time in a planning month for a flight:

```
%leg_month_block_time% =
   overlap(%month_start%,
           %month_end%,
           departure,
           arrival);
```

Note: All arguments are either RelTime or AbsTime!
The result is RelTime
If there is no overlap at all, 0:00 will be returned.

Built-in functions

Scale time

```
scale_time
```

used to scale time intervals differently during a day (24:00)

```
scale_time(act_start, act_end, default_scale,
           start1, end1, scale1,...)
```

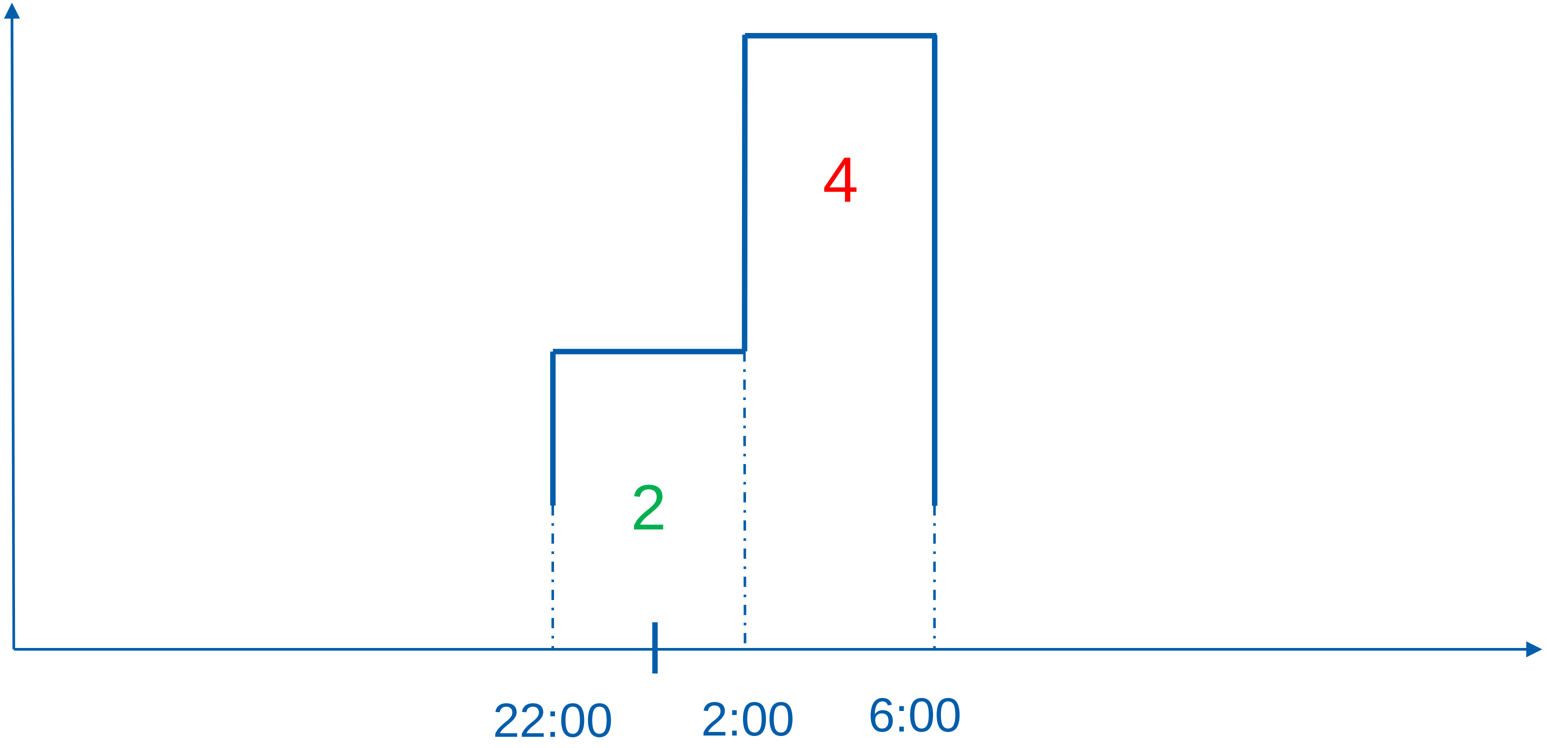
The function returns a `reltime` value that is a scaled number of hours that overlaps an absolute time interval

Built-in functions

Scale time

At night, crew is awarded extra credit time,
double between 22 and 02 and 4 times between 02 and 06

`scale_time`

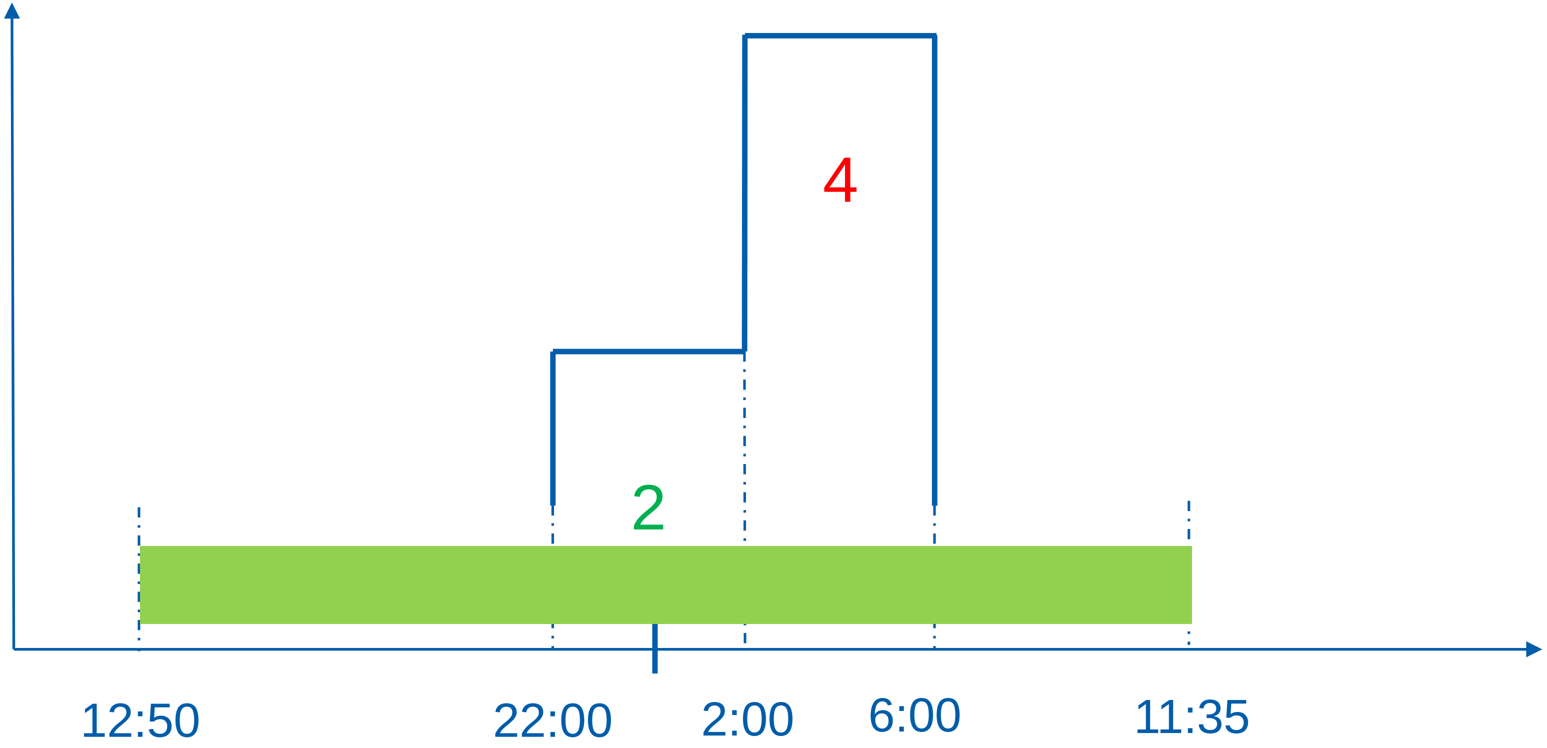


Built-in functions

Scale time

What would the credit time be for an activity from 12:50 to 11:35?

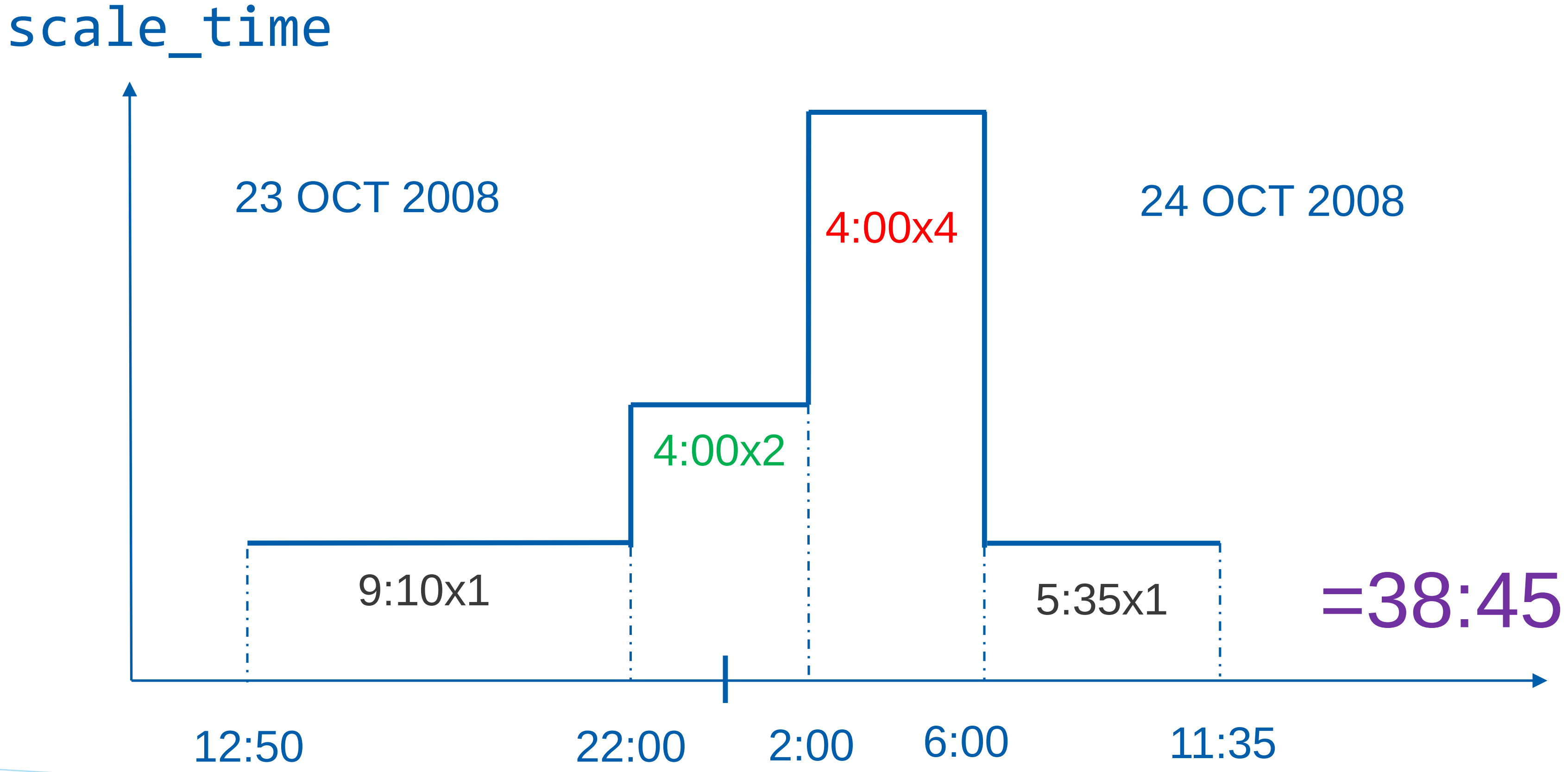
scale_time



Built-in functions

Scale time

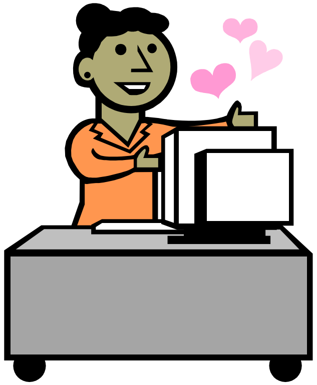
```
%credit_total% =
    scale_time(23oct2008 12:50,24oct2008 11:35,1,
               22:00,02:00,2, 02:00,06:00,4);
```



Exercise 4

~75 mins

Exercise 4 summary



Chapter 5

If-then-else

Tables

If-then-else

Sometimes you need to use different values depending on the outcome of certain calculations

Example:

A deadhead flight costs more with a different fleet,
and it is very expensive with another company (OAG).

If-then-else

The deadhead cost with another company (OAG) is 1000 and with another fleet 500. Otherwise it is only 400

If flying OAG then 1000,
if another fleet then 500
else 400.

If-then-else

```
%cost_of_deadhead% =
    if oag then 1000
    else if %other_fleet% then 500
    else 400;
```

If-then-else

Hotel cost expressed with an if-then-else definition:

```
%hotel_cost% =
    if %hotel% = ”jerrys inn” then 25
    else if %hotel% = ”holiday inn” then 350
    else if %hotel% = ”sheraton” then 500
    else if %hotel% = ”plaza” then 650
    else 999;
```

Tables

- Tables are a convenient way of expressing complex if-else statements

- As an example, let us look at the hotel cost expressed with a table instead.

Tables

Hotel cost expressed as a table:

```
table hotel_cost_tab =
    %hotel%          -> %hotel_cost%;
    ”jerrys inn”     -> 250;
    ”holiday inn”    -> 350;
    ”sheraton”       -> 500;
    ”plaza”          -> 650;
    - -> 999;
end
```

Tables

You want to award crew points depending on the length of the flying time and if it is a deadhead or not

Flying time

- less than or equal to 1h gives 3pt
- 1h to 3h gives 5pts
- over 3h gives 7pts.

Deadhead flights give one point less.

Tables

```
table awarded_points_tab =
   %flying_time%, deadhead -> %awarded_points%;

   <= 1:00,     false -> 3;
   )1:00, 3:00),false -> 5;
   > 3:00,      false -> 7;
   <= 1:00,     true  -> 2;
   )1:00, 3:00),true  -> 4;
   > 3:00,      true  -> 6;
   -,- -> 999;
end
```

Tables

As with functions, it is possible to pass an argument to a table. In this way, you may alter the outcome of a table depending on the argument

For example, you may need the type of a day in either Swedish or English.

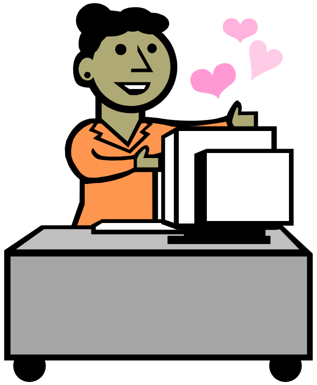
Tables

```
table week_day_tab(string language) =
    language, %week_day_number%
        -> %week_day_type_string%;
    “UK”, (1,5) -> “Weekday”;
    “UK”, (6,7) -> “Weekend”;
    “SE”, (1,5) -> “Veckodag”;
    “SE”, (6,7) -> “Helg”;
    -,- -> “Unknown”;
end

%week_day_number% = 4;
%week_day_type_string%(“SE”) ➔ “Veckodag”
```

Exercise 5

~45 mins

Exercise 5 summary



Chapter 6

External Tables

Set

External Tables

- External tables make it possible to interact with data in files or data base (no difference from Rave point of view)
- Data may come from other systems
- Data may be changed instantly by planners
- Data does not have to be known at compilation
- Table definition in Rave code + external data file
- External tables are also called Etables or etabs (dtables).

External Tables

Data file

- The information (data) in an Etable is saved in a separate text file
- The data is ordered in rows and columns
- There is a header describing the columns

See
Man page (unix) for a complete syntax description

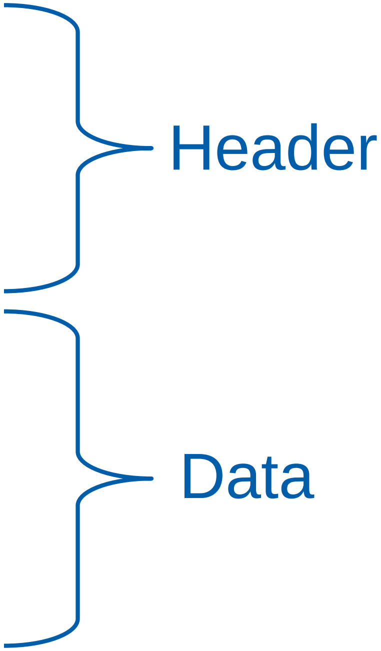
See
Rave Reference > Definitions > Tables
for more information about External Tables

External Tables

Example

Data file named: `AircraftFamilies.etab:`

```
2
Sac_type “Aircraft Type”,
Sac_family “Aircraft Family”,

“747”, “747”,
“74E”, “747”,
“727”, “727”,
“72X”, “727”,
```



Header

Data

External Tables
Table definition

The external table definition is very much like a normal internal table:

```
table aircraft_family_tab =
    aircraft_type -> String %aircraft_family%;
    external ”AircraftFamilies.etab”;
    ac_type -> ac_family;
    - -> ”No family”;
end
```

File name with (or without) suffix `.etab`
May be a variable

External Tables
Table definition

This means:

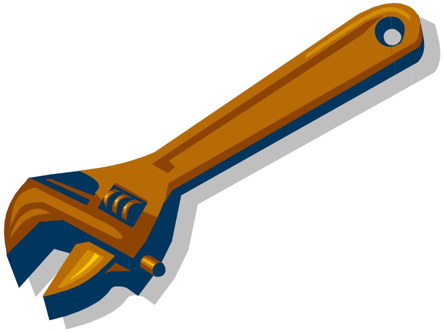
- take the value of the keyword `aircraft_type`
- open the file `AircraftFamilies.etab`
- search in the column `ac_type` for that key
- if you find a match:
  - return the value from the column *`ac_family`*
- if you don’t find a match:
  - return “No family”
- … for the definition `%aircraft_family%`.

External Tables

Table Editor

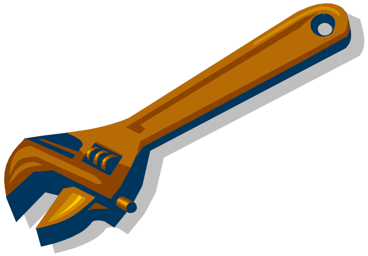
Let’s look at the Table Editor:

- View and Edit data
- Configure etable layout.



Demonstration

Table Editor



Tables
Multiple results

Internal and external tables can return multiple results:

```
table hotel_cost_tab =
    %hotel%      -> %hotel_cost%,
                    %crew_likes_hotel%;
    “jerrys inn” -> 250, false;
    “holiday inn”-> 350, true;
    - -> 999, false;
end
```

Defines both `%hotel_cost%` and `%crew_likes_hotel%`.

Tables

Multiple results

- It is more efficient to look up several values at the same time

- Only possible when they use the same key
- Different keys need to be implemented in separate tables

- Multiple table definitions may use the same external file

- Not all columns in an external file need to be used.

Virtual columns

- The virtual column `row_number` returns the row number for *all* rows in an external table
- The virtual column `match_number` calculates the row number for *matching* rows in an external table

- If the key matches multiple rows in the table, only the first row will be returned. Therefore the order of rows in etables is important. Dtables are by default unordered

- It is possible to traverse all matching rows.

Sets

A set:

- is a group of items
- all items have the same data type
- may be parameterized
- may be external.

It is only possible to check if an item is part of a set.

Sets
In

You have a group of Asian airports, and want to know if you arrive at one of them:

```
set asian_airports = parameter
    “BKK”, ”SIN”, ”HKG”, ”PEK”, ”NRT”
    remark ”Asian airports: ”;

%is_asian_landing% =
    arrival_airport_name in asian_airports;
```

Sets

Concatenation

When defining the set, do not forget the ‘ , ’:

```
set asian_airports = parameter
    “BKK” ”SIN”;
```

➔ results in “BKKSIN”

This is true for all string handling:

```
%string% =
    “my long string”
    “that does not fit into one row...”;
```

➔ “my long stringthat does not fit into one row...”

External Sets

The external set is defined by reading all entries of a external table column:

```
set asian_airports =
    external string
    “asian_airports.etab”.“ap”;
```

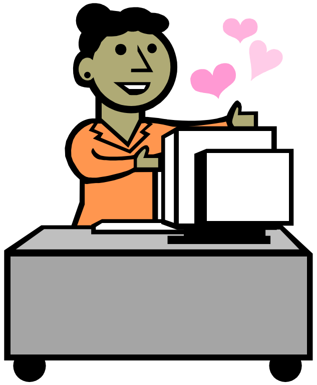
Will create a set which contains all airports listed in the column “ap” of the external table “`asian_airports`”.

External sets provide no functionality which cannot be implemented with the use of the ordinary external table syntax. External sets may lead to nicer code though.

Exercise 6

~75 mins

Exercise 6 summary



Chapter 7

Rave level definitions

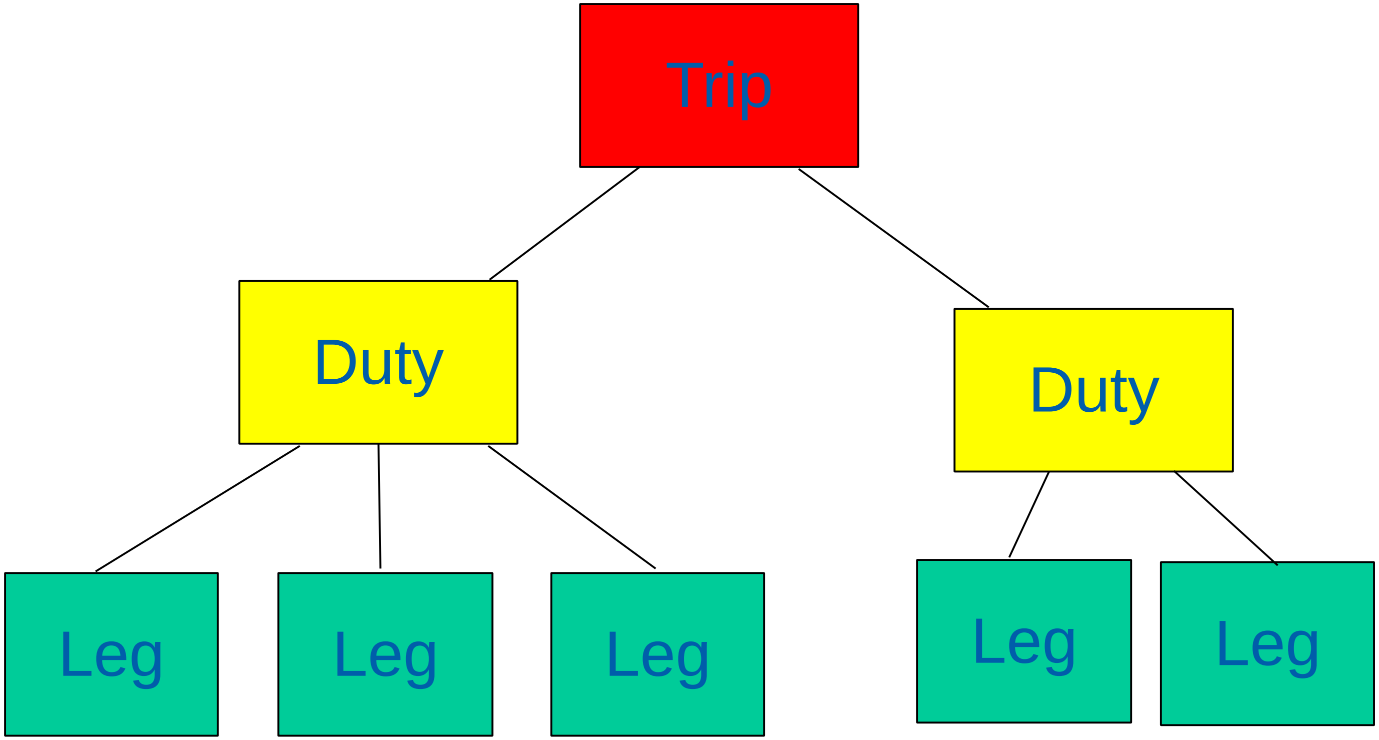
Level dependencies

Levels

- Levels group legs or other levels in a chain that have something in common
  (same working day/period/…)
- Levels give structure and speed up calculation
- There are always two basic levels:
  `atom` (smallest) and full `chain` (largest)
- Between the two basic levels, intermediate levels (`duty`, `trip`,...) can be defined
- Intermediate (non basic!) levels are defined in terms of already existing levels.

- Note:
- The levels defined in Rave may differ in meaning from the views in Studio
  (show Rosters, Trips, Legs, etc.)
- There is no technical relation between levels and names given to module files.

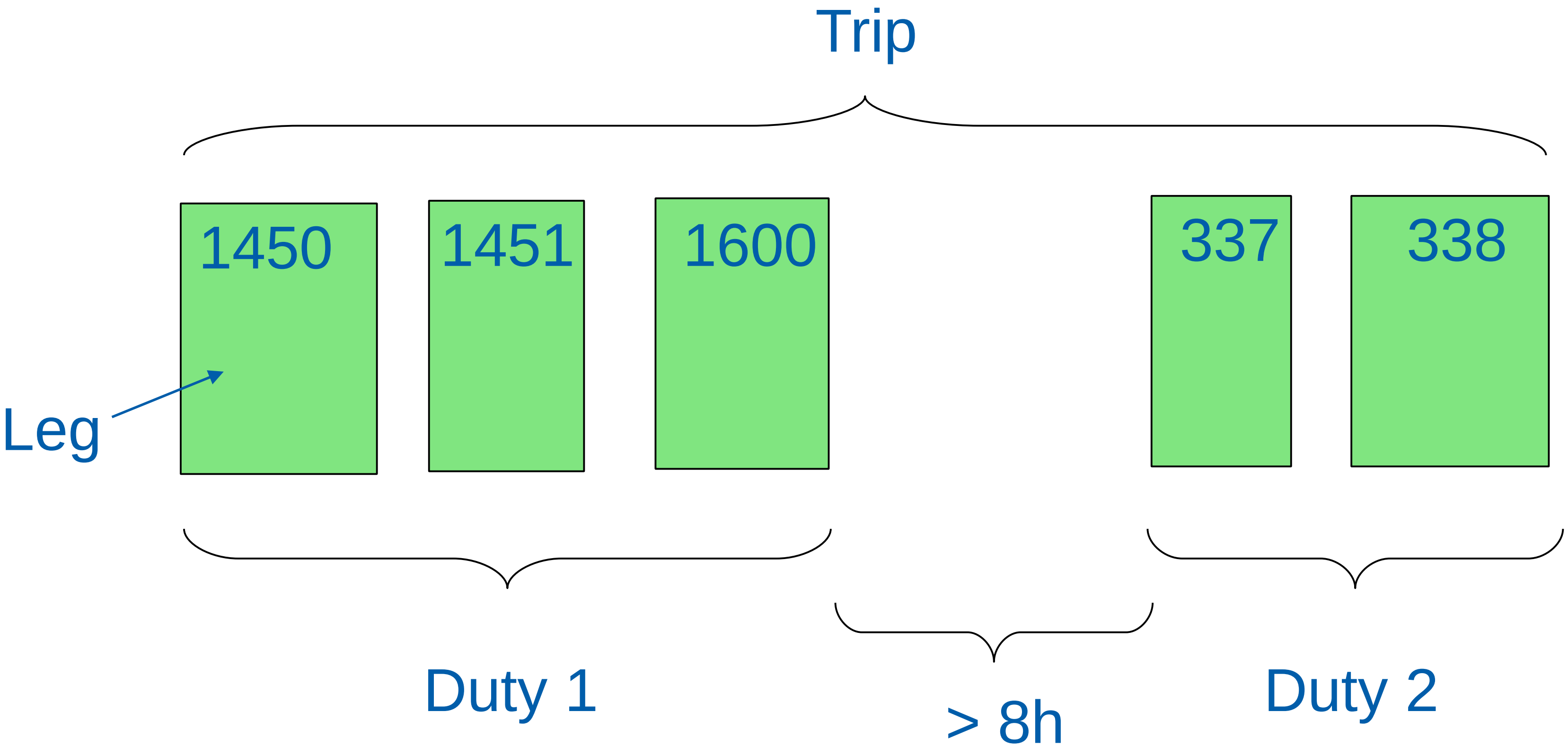
Often defined levels



Levels

Examples

Possible to work only on a part of a chain (sub-chain) in an efficient way in Rave



Levels

Leg

We need a ‘beginning’ for the level definitions
Rave has a built in level ‘atom’ that is defined with a special notation:

Every object is last in a leg sequence

```
module levels
…
level leg =
    when(true);
end
```

Levels
Duty

Levels are normally defined in terms of already defined smaller levels.
Example: There is a new duty when the *leg* connection time is large enough:

”A leg is last in a duty when the connection time > 8:00”

```
module levels
…
level duty =
    is_last(leg)
    when(%leg_connection_time% >= 8:00);
end
```

Levels
Chain

Rave has an other built in level ‘chain’ which contains all objects
No object is ever last in a chain sequence

```
module levels

…

level trip =
    when(false);

end
```

Note: For Rostering, replace `trip` with `roster`.

Levels

Dependencies for keywords

ATOM

- The smallest object (usually a leg)
- Most keywords are atomic. Examples: `departure, arrival, arrival_airport_name, flight_number, deadhead`

CHAIN

- The full chain
- Keywords with only one value for the full chain have this level
  Examples: `crr_crew_id,  homebase`

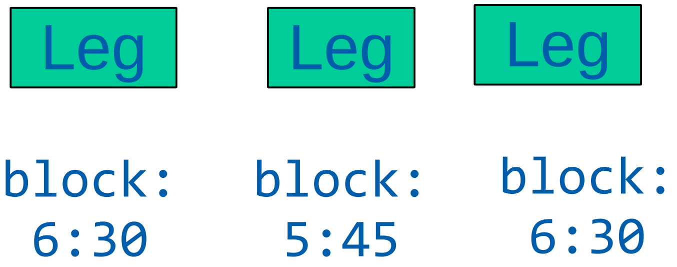
CONST

- Keywords with the same value for all objects in the planning environment have this level
  Examples: `is_cas_system, user`

Levels

Variable Dependencies

- Every expression in Rave has a level dependency
- A unique instance of the rule or value exists for each object on this level
- Example:
  The variable `%block_time%` is leg dependent,
  each leg will have its own (instance) value for this variable



Leg Leg Leg

block: 6:30

block: 5:45

block: 6:30

Levels

Duty dependent

- A variable that is duty dependent is evaluated for the duty that the active leg belongs to

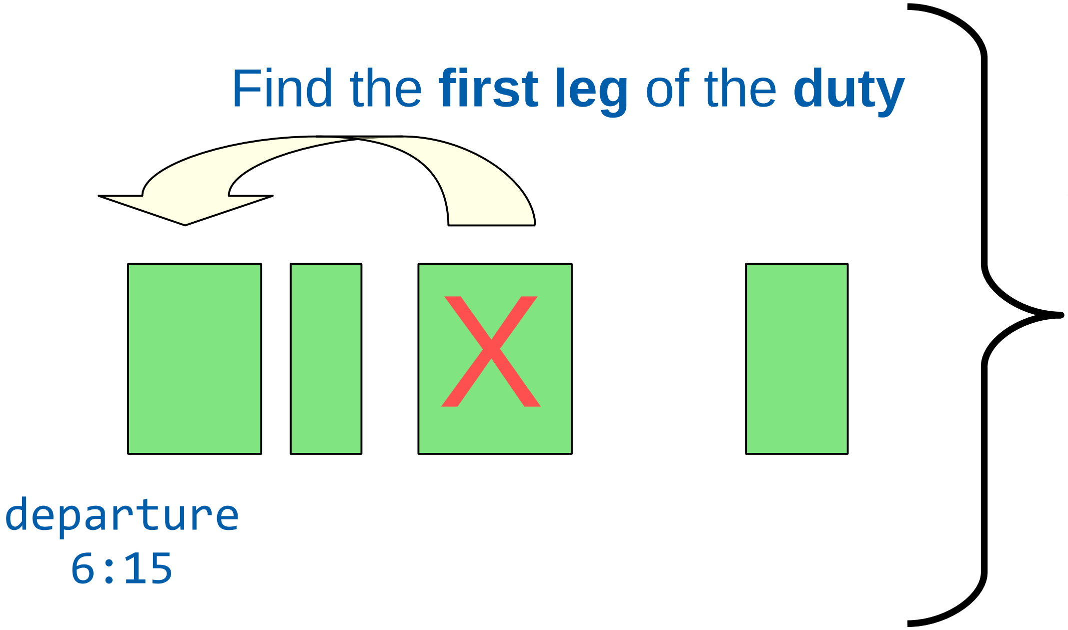
- When a variable is duty dependent, it will return the same value (the same instance) for all legs inside the duty.

Levels

Duty dependent

```
%duty_departure% =
    <departure of first leg in the duty>;
```

Find the **first leg** of the duty

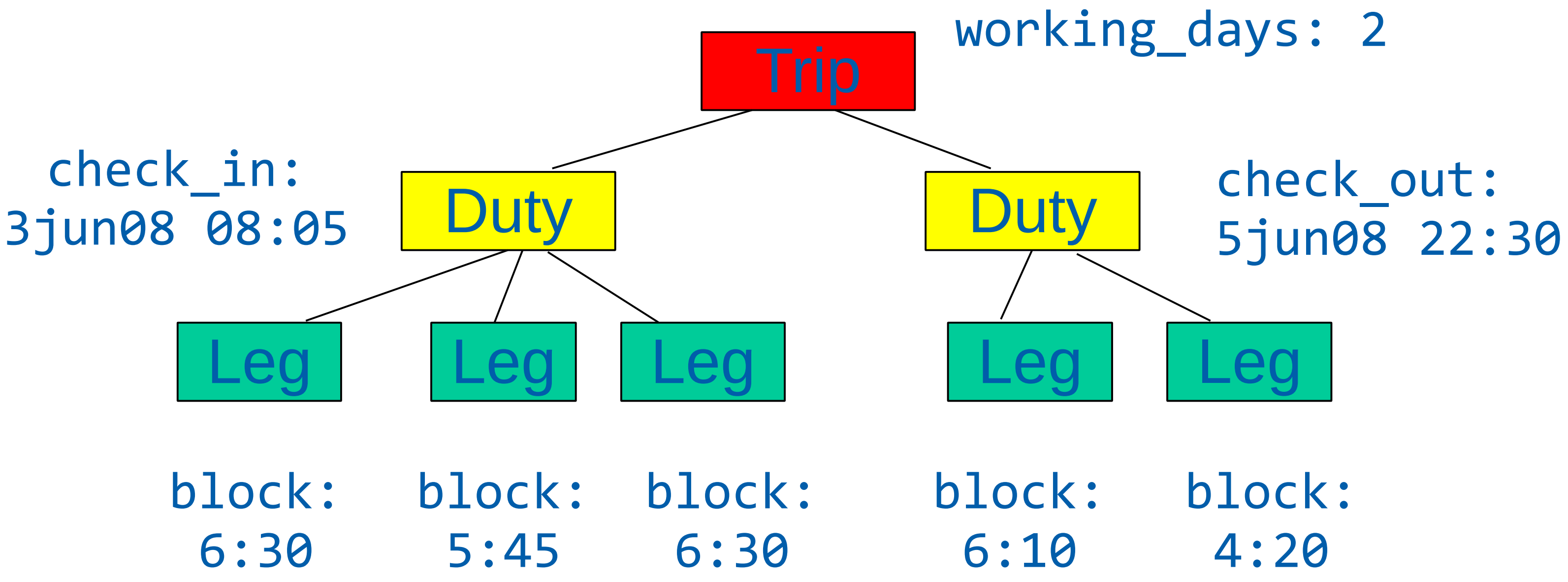


departure
6:15

%duty_departure% = 6:15

Levels

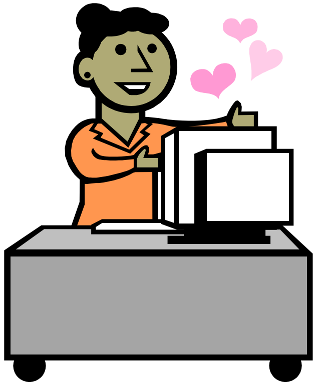
Dependency



Exercise 7

~30 mins

Exercise 7 summary



Chapter 8

Traversers

void values

Filters

Traversers

Traversers* are used to find values on legs other than the current one

Departure time of a duty is evaluated as the departure of the first leg in the duty:

```
%duty_departure% =
    first(leg(duty), departure);
```

* Traversing = moving across, from one to another place

Traversers

Rave Traversers

sum, avg, max, min, any, all, **count**, cat

first (earliest), last (latest), next, prev

is_first, is_last

Note: bold traversers do not take any expression as argument

Traversers

Multiple Data Types

Some traversers can be used with multiple data types:

Traverser	Valid Data Types
sum, avg	RelTime, Int
min, max	AbsTime, RelTime, Int

Traversers

Examples

```
%duty_block_time% =
    sum(leg(duty), %block_time%);

%trip_has_deadhead% =
    any(leg(trip), deadhead);
```

Traversers

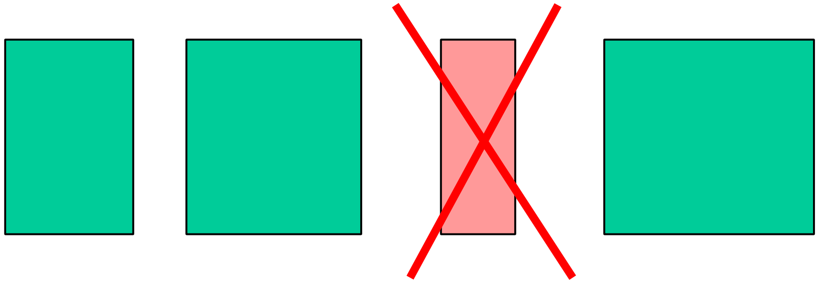
Examples

```
%last_dh_home% =
    deadhead and is_last(leg(trip));

%total_time_away% =
    last(leg(trip), arrival)
    - first(leg(trip), departure);
```

Traversers
where

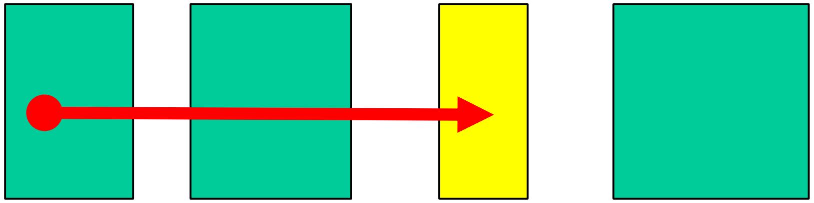
```
%active_block_time% =
    sum(leg(trip), %block_time%)
    where(not deadhead);
```

The where expression must be put in parenthesis.



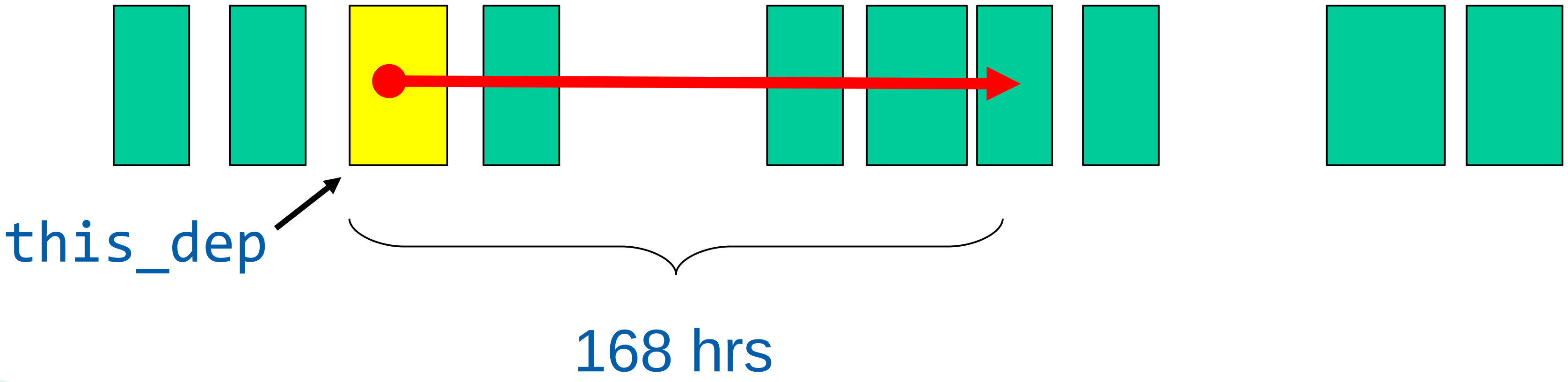
Traversers
from/to

```
%accumulated_block_time% =
    sum(leg(trip), %block_time%)
    from(first)
    to(current);
```



Traversers
while

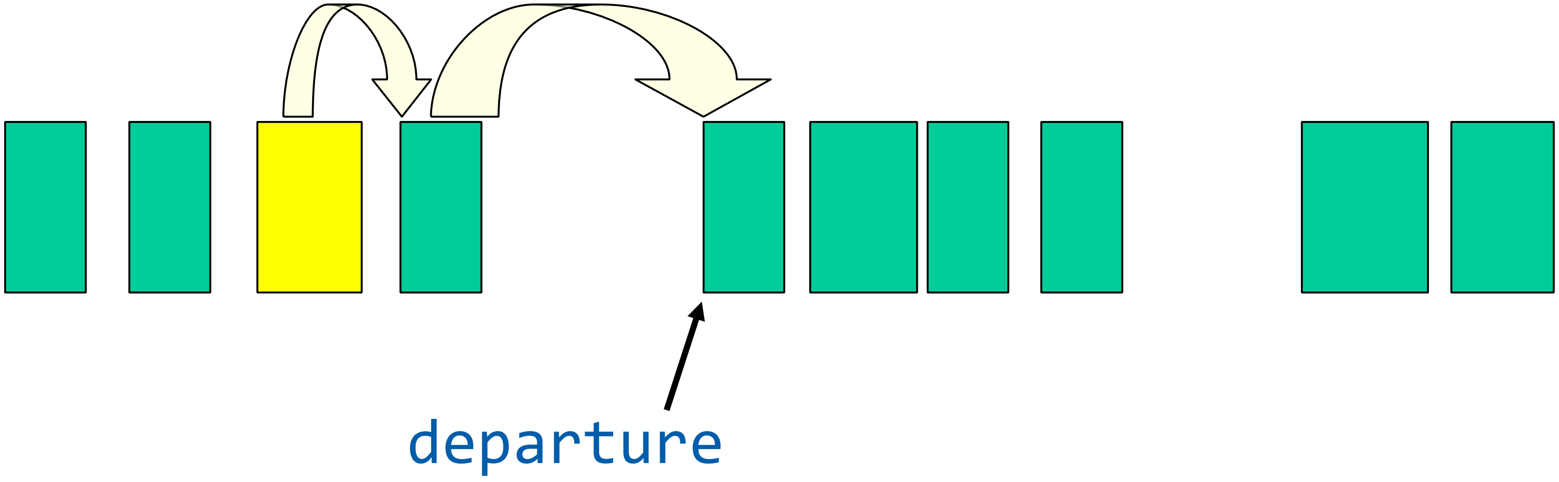
```
%block_time_168hrs_fwd% =
    let this_dep = departure;
    sum(leg(roster), %block_time%)
    from (current)
    while (departure <= this_dep+168:00);
```



Traversers

Nested

```
%second_following_leg_departure% =
    next(leg(trip),
        next(leg(trip), departure));
```

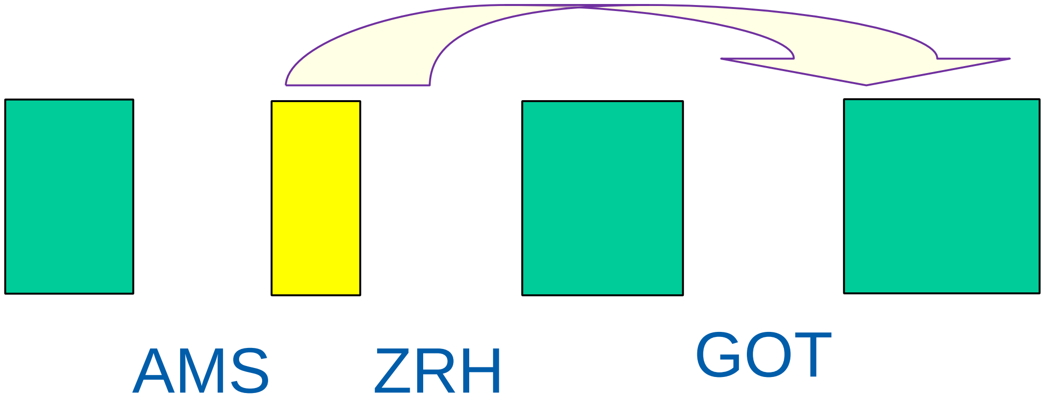


Traversers

Sliding

```
%first_following_departure_from_got% =
    next(leg(trip), departure)
    where (departure_airport_name
            = “GOT”);
```

Finds next possible leg that fulfils the condition



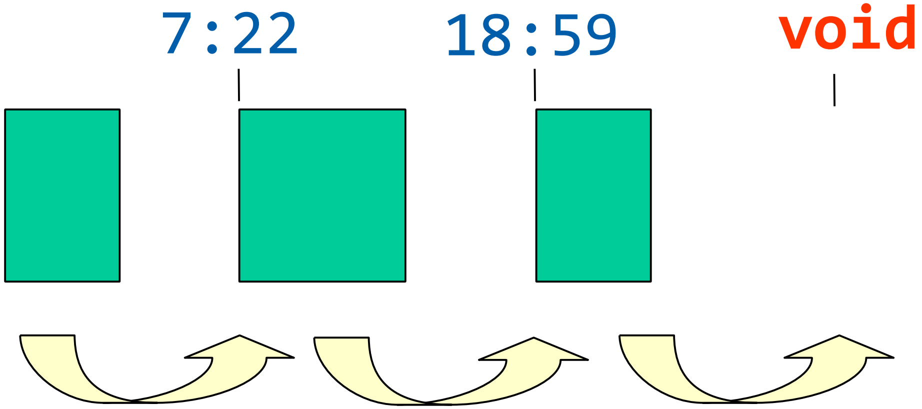
Void values

Calculate connection time between two legs:

```
%connection_time% =
    next(leg(duty), departure) - arrival;


%connection_ok% =
    %connection_time% >= 0:25;
```

Void values

```
next(leg(duty), departure)
```



will return ’void’ for the last leg in a duty

Void values
default

`void(expr)` returns `true` if `expr` is void

```
default(expr1, expr2)
```

returns result of `expr1` if `expr1` is not void, otherwise returns result of `expr2`

Example:

```
%cxn_ok_not_void% =
    default(%cxn_time% >= 0:25, true)
```

functions

- There are constant void reserved words: `void_string`, `void_abstime`, `void_reltime`, `void_bool` and `void_int`
- Used when a value for a Rave variable can not be defined
- Often used in if-then-else and table expressions

Void values

Propagated

- Void values are propagated through the code until they are ‘caught’ by a `void()` or `default()` expression
- If not caught, `void` is returned as result value
- If `void` is passed as argument to a function, it will *immediately* return void
- If local ‘let’ variables evaluate to `void`, the definition will *immediately* return void.

Filters

- **almost never used**
- **are similar to levels**
- **are defined as subsets of levels**
- **could be used instead of `where` clause.**
- **Syntax:**

```
[global export]
filter NAME = LEVELNAME(condition);
```

Filters
Example

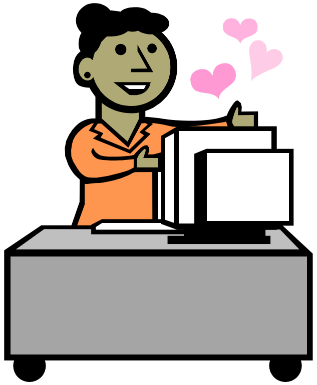
To count all active legs in a trip:

```
module trip

/* with where */
%num_of_active_legs% =
    count(leg(trip)) where(not deadhead);

/* with filter */
filter active_legs = leg(not deadhead);
%num_of_active_legs% =
    count(active_legs(trip));
```

Exercise 8

~60 mins

Exercise 8 summary



Chapter 9

Rule sets
Parameters
Source Code
Modules
DWS

Rule sets

- Compiled set of rule definitions
- Binary file used by the system
- May be loaded into an application (Studio, optimizer,...)
- Also called rule package

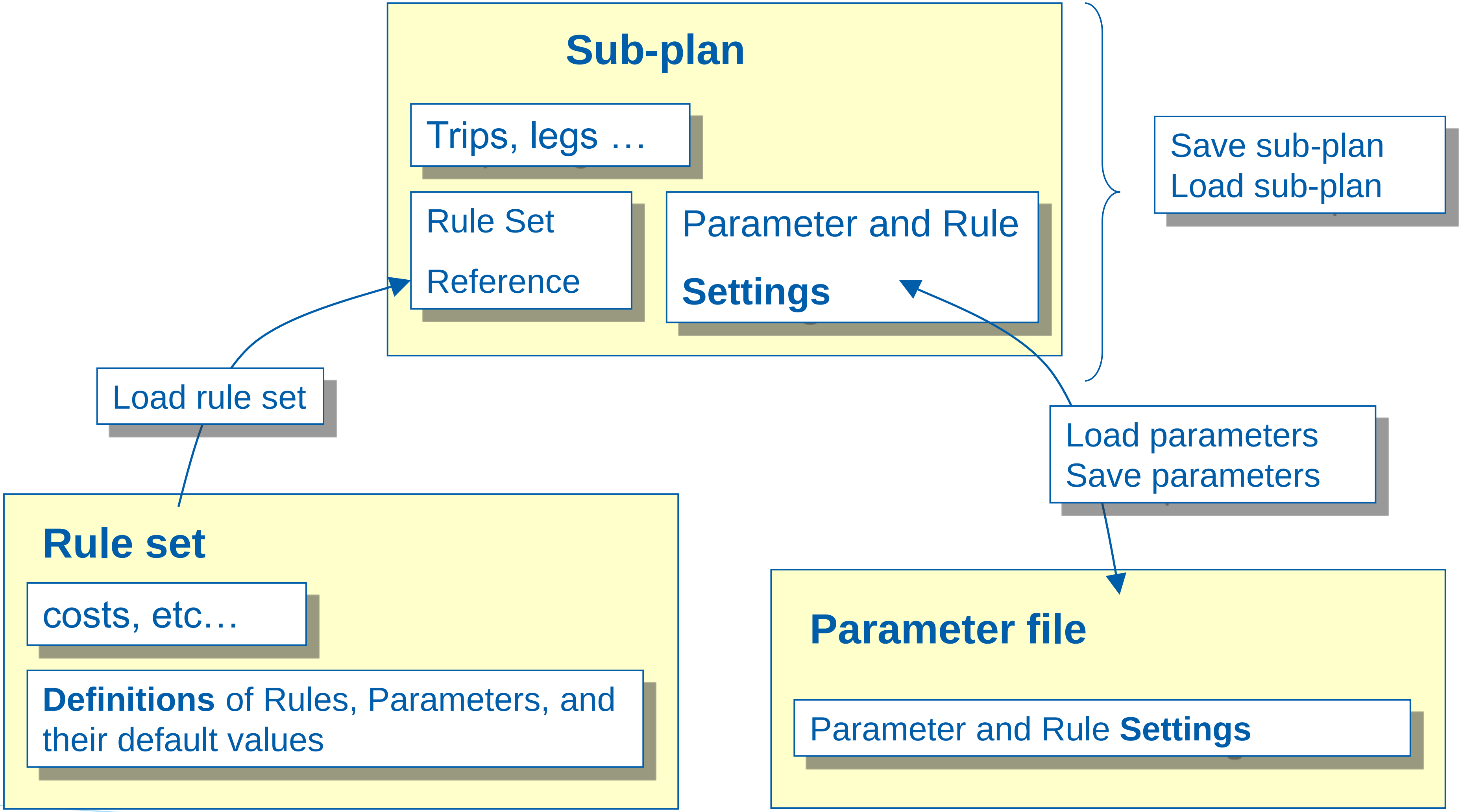
Sub-plans, parameter files and rule sets

- A rule set defines a set of rules, parameters and their default values

- A sub-plan file contains a reference to a rule set and current settings of rules and parameters. The settings are saved with the sub-plan

- You can save the current settings to a separate parameter file. These settings may be loaded into another sub-plan.

Sub-plans, parameter files and rule sets



Sub-plan

Trips, legs …

Rule Set

Reference

Parameter and Rule

Settings

Save sub-plan
Load sub-plan

Load rule set

Load parameters
Save parameters

Rule set

costs, etc…

Definitions of Rules, Parameters, and their default values

Parameter file

Parameter and Rule **Settings**

Rule source code

- Separated into several files
- Possible to share code between rule sets
- The compiler searches for files in different directories*

* Directories and search order may be reconfigured using Carmen resources.

Rule source code

Structure

Default CARMUSR structure:

```
$CARMUSR/
          …
          crc/
               source/
                    RaveI_ruleset
               require/
               report/
               modules/
                         ravei_exercises
                         trip_rules
                         trip_cost
```

Rave repository

Modules

- Rave definitions grouped by functionality

- A variable in a module is usually accessed by

  `[module_name] . [variable_name]`
  Example: `cost.%of_deadhead%`

- Variables not inside a module end up in the special module `_topmodule`

Modules

- Encapsulation of definitions:
  - namespace
  - visibility.
- Reuse in multiple rule sets
- Inheritance (covered in the Rave II course).

Modules
`import/export`

- All definitions to be used by other modules must be *exported*

- A module must *import* all other modules it needs definitions from

- The modules are located in:
  …`/crc/modules/file_name`

- and the file starts with "`module file_name`".

Modules

import/export

```
File: $CARMUSR/crc/modules/duty:
module duty
import leg;
…
export %paid_time% =
    sum(leg(duty), leg.%paid_time%);
```

This code creates the variable `duty.%paid_time%,` which can be used by other modules if they `import` the module “`duty`”.

Modules

global export

Common definitions can be *globally* exported and used without namespace reference. There is no functional change, only less to write.

```
import levels
sum(leg(duty), …)
```

instead of:

```
import levels;
sum(levels.leg(levels.duty),…)
```

Note: Scripts and reports always need to use `module.var_name`

Modules
global export

Common definitions are globally exported and used without namespace reference

Example:

```
module levels
global export level leg =
    …
global export level duty =
    is_last(leg)
     when(%new_duty%);
end
```

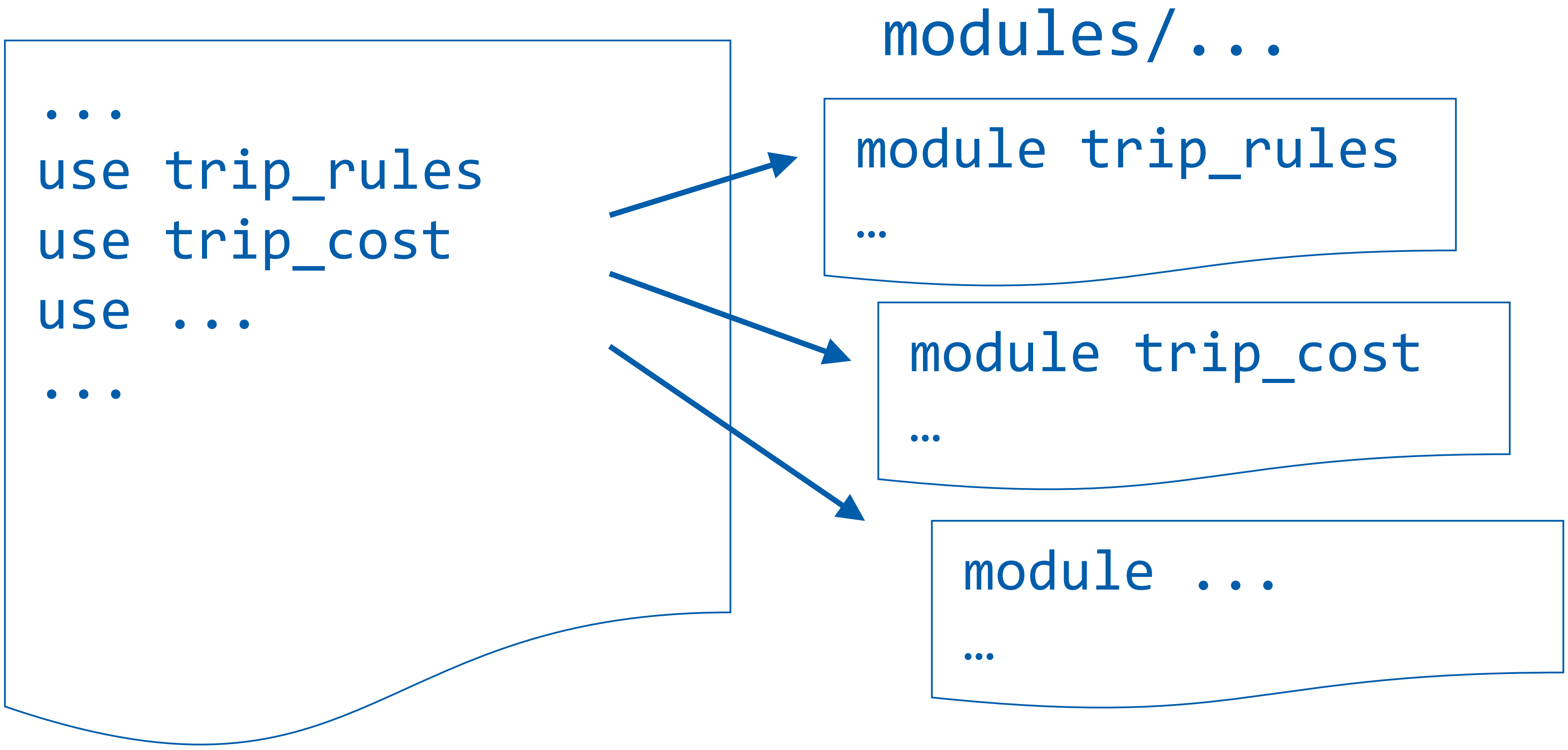
source

```
source/<top file>
```

- Top file for each rule set
- May be compiled
- Mainly contains “use” and “require” statements
- May contain Rave definitions.

Source files

`source/rule_set_file`



```
...
use trip_rules
use trip_cost
use ...
...
```

`modules/...`

```
module trip_rules
…
```

```
module trip_cost
…
```

```
module ...
…
```

Separation of code between applications

Define modules only used by Studio:

source/rule_set_file

```
...
use trip_rules
#if product(Studio)
   use studio_code;
end
...
```

modules/studio_code

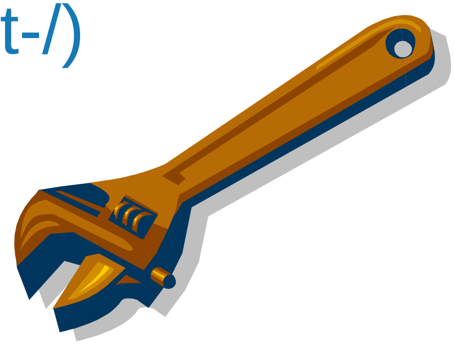
```
module studio_code
…
```

Note: Available products are set the by the resources *<appliation>.RaveCompile.CcrcAttributes*
e.g. `gpc.RaveCompileCcrcAttributes : product=gpc,studio`

DWS demonstration

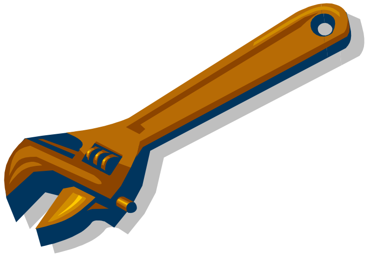
Let’s look at some more DWS features:

- Detach windows
- Double click to maximize (editor) window
- Tooltip for variable dependency, range and reference
- Configuring available products
- Shortcut keys (F3, Ctrl-Shift-F3)
- Auto complete (Ctrl-Space, Alt-Shift-/)
- Open Rave Def. (Ctrl-Shift-D)
- Refresh
- Clean



Demonstration

DWS



Require statements

Include code from other files:

```
require FILE
```

Included files may be different depending on application:

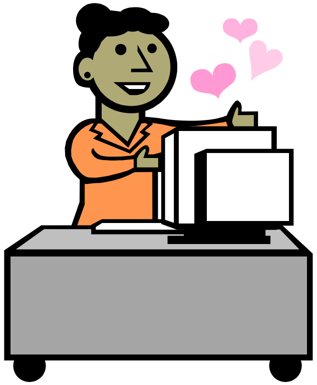
- Studio: search for `FILE` first in the `report/` and then in the `require/` directory
- Optimizer: search only in the `require/` directory (obsolete)

Note: Studio still uses require files for rudob and other map variables.

Exercise 9

~60 mins

Exercise 9 summary



RULE BOOK

Rules

Rules:

- are used to check legality
- have level dependencies
- are always considered
  - Studio
  - optimizer
- may be turned on/off by planner.

Rules

Example

Restrict APC from building inefficient trips:

”For short-haul trips, the maximum time between 2 legs in a duty is 4 hours”

Variable:	calculate the *actual* time between 2 legs
Parameter:	min or max *limit* value of a rule
Rule:	*comparing* the variable to the parameter – result must be Boolean

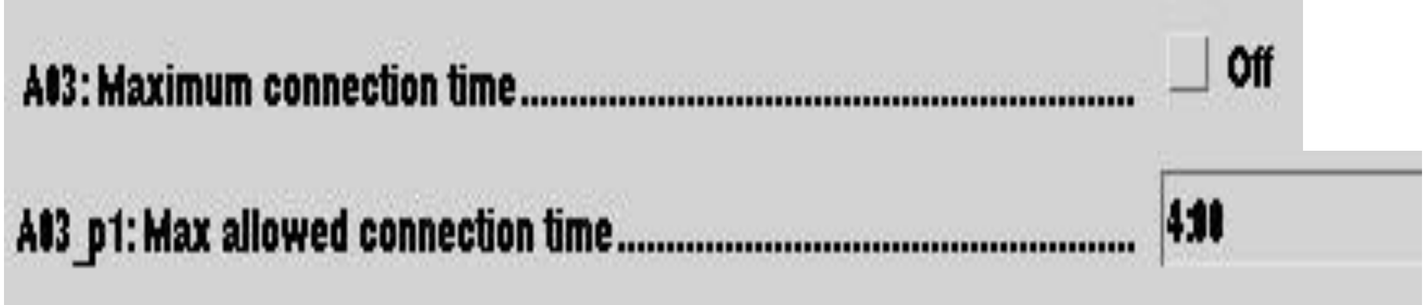
Rules

Example

```
module rule_exp
rule (off) max_cnx_time_rule =
    valid leg.%short_haul%;
    leg.%cnx_time% <= %max_cnx_time_p%;
    remark “Exp03: Maximum connection time”;
end
%max_cnx_time_p% = parameter 4:00
     remark “Exp03_p1: Max connection time”;
  ---------------------------------------------------------------------

module leg
export %cnx_time% = next(leg(duty), departure) – arrival;
export %short_haul% = not %long_haul%;…
```

Rules

Example



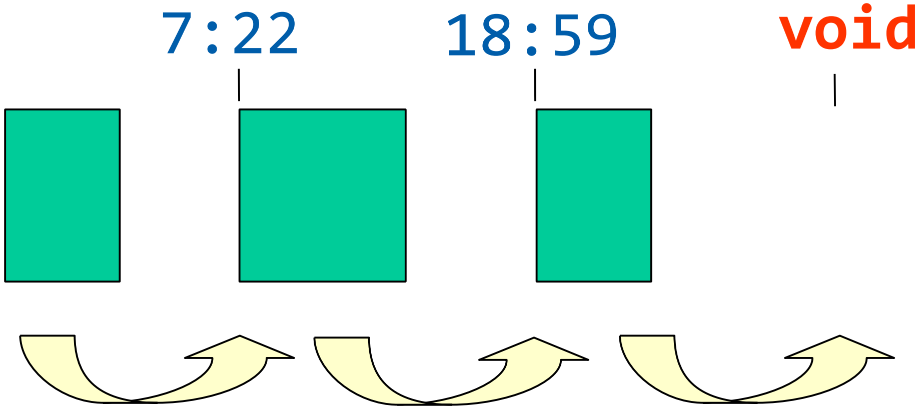
A03: Maximum connection time.. ☐ Off

A03_p1: Max allowed connection time.. 4:30

This rule must be turned on by the planner,
it will only be used on short-haul legs (valid statement).

Turn on: `Options > Preferences > Show Children`

Rules
void



`next(leg(duty), ...)` might return `void`

Rule expressions returning void
are always considered legal.

Rules

void

```
rule r =
    valid %v%;
    %c%;
end
```

Outcome of rule ”r”		%v%		
		TRUE	FALSE	VOID
%c%	TRUE	**LEGAL**	disregard	disregard
	FALSE	**ILLEGAL**	disregard	disregard
	VOID	**LEGAL**	disregard	disregard

%v% and %c% should have the same dependency.

This will be the dependency of the rule.

Rules

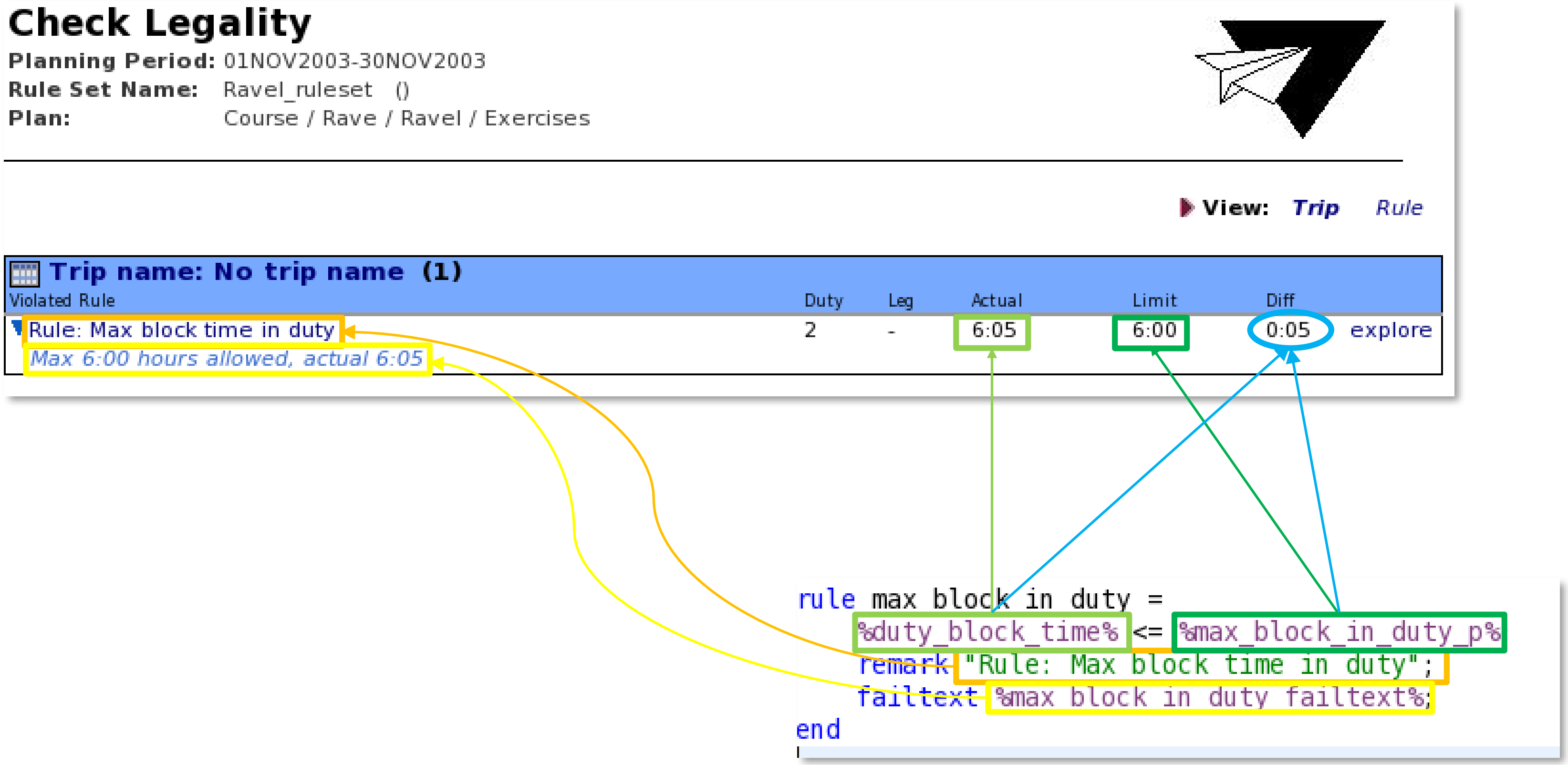
Remarks

For remarks, there are different labels:

`remark` and `planner`:

```
rule rule_name =
    %actual_value% <= %limit_value%;
     remark “This text is shown in the params form”,
     planner “The planner description is added ”
             “automatically to the online help ”
             “that may be viewed by pressing ”
             “‘F1’ key”;
end
```

The Check Legality Report



Check Legality

Planning Period: 01NOV2003-30NOV2003
Rule Set Name: Ravel_ruleset ()
Plan: Course / Rave / Ravel / Exercises

View: *Trip* *Rule*

Trip name: No trip name (1)

Violated Rule	Duty	Leg	Actual	Limit	Diff	
Rule: Max block time in duty *Max 6:00 hours allowed, actual 6:05*	2	-	6:05	6:00	0:05	explore

```
rule max block in duty =
    %duty_block_time% <= %max_block_in_duty_p%
    remark "Rule: Max block time in duty";
    failtext %max block in duty failtext%;
end
```

Rules

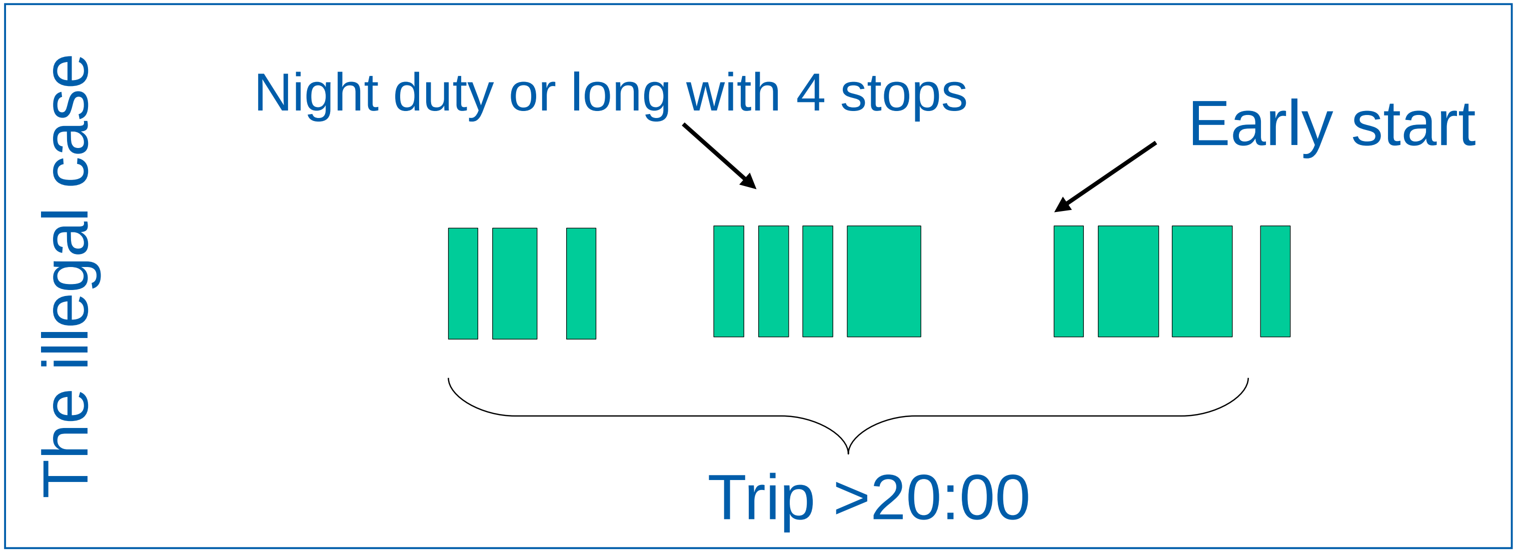
Example

To make sure pilots have enough time for night rest there is a rule for early starts:

A duty may not start before 9:00 after a night duty or after a long duty (>8:00) with 4 or more stops. There is no limitation on trips that are shorter than 20:00.

Example

- First we draw a picture of an illegal trip:



- Then we decide which duty will be illegal:
  - Here, it is more natural to consider the last duty as illegal
  - Decided on a rule by rule basis.

Rules

Example

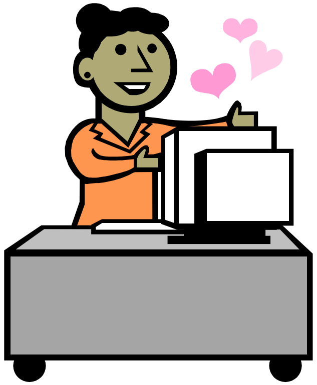
```
rule no_early_start_after_night =
   valid %trip_length% > 20:00
         and not is_first(duty(trip))
         and %check_in% < 9:00;

  not prev(duty(trip),
     %is_night_duty%
     or %long_with_4_stops%);
end
```

Note: Normally ‘pure’ variables are used for report and debugging purposes.

Exercise 10

~120 mins

Exercise 10 summary



Chapter 11

Costs

Cost function

Map Variables

Costs

The cost function:

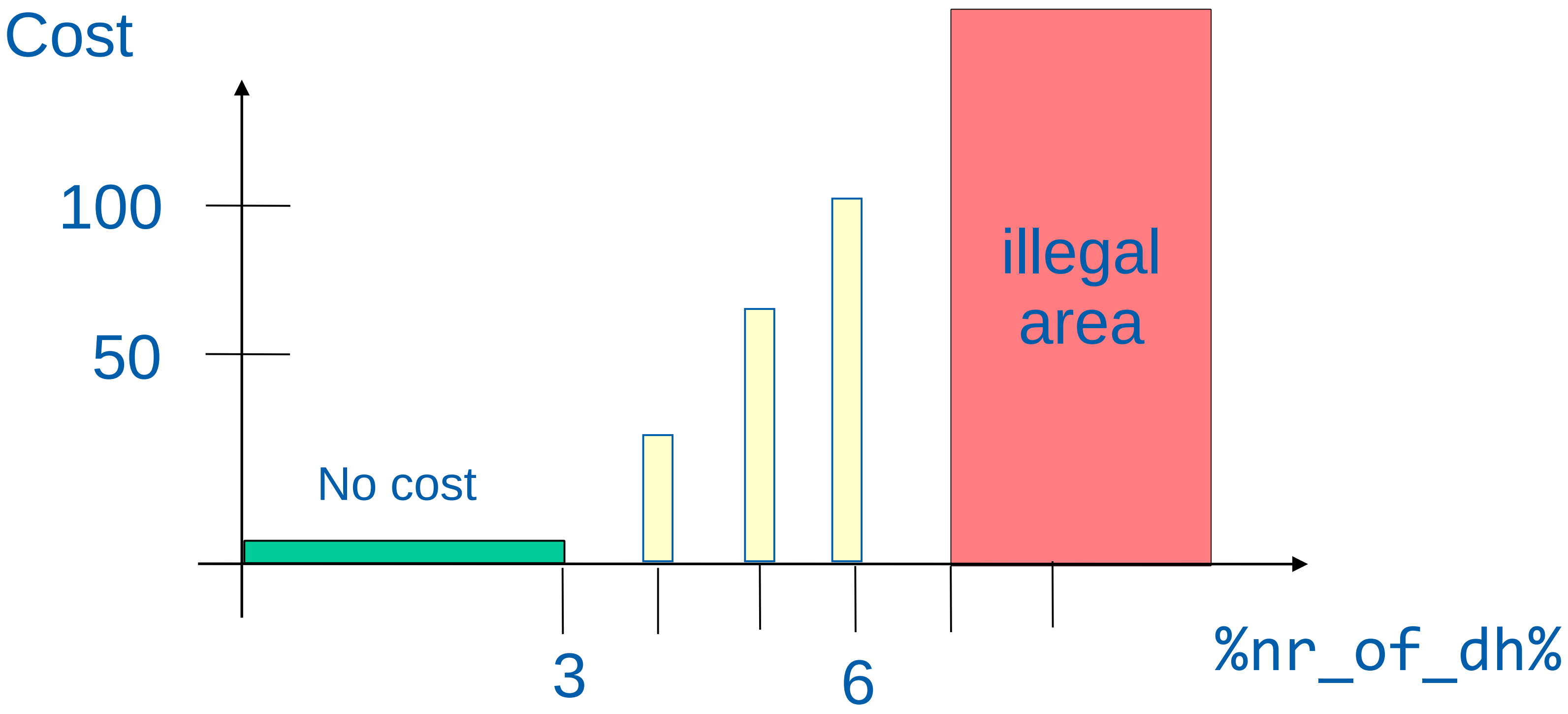
- defines optimization objective
- used to guide the optimizer towards better solutions
- used to avoid events/patterns that are legal but undesirable
- costs can be used to loosen up legality.

Example

Limit the number of deadheads in a trip:

- no cost for 1, 2 or 3 deadheads
- increasing cost for 4, 5 and 6 deadheads
- 7, 8 and 9… deadheads are illegal.

Cost
Example



Cost

Example

```
module rule_exp
import trip;
rule max_deadheads_in_trip =
    trip.%nr_of_dh% <= 6;
end
```

```
module cost
import trip;
export %of_deadheads_in_trip% =
    nmax(trip.%nr_of_dh% - 3,0) * 33;

export %of_trip% =
    … + %of_deadheads_in_trip%;
```

Map Variables

- A map variable is a Rave dictionary variable that is known to a Jeppesen application, such as Studio or APC
- Map variables are imported to the system kernel from Rave, and the value is usually (with a few exceptions) the name of another Rave variable

- The system kernel finds the value in two steps:
  - Ask Rave for the value of the map variable
  - Ask Rave to calculate the value of the variable whose name was obtained in step 1
- The names of the map variables must not be changed as this would make them unrecognizable to the Jeppesen application.

Cost

Map Variables

Example:

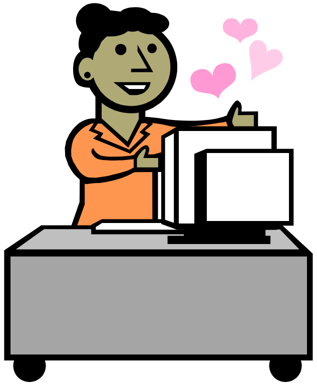
Set the rave variable that should be used by pairing APC to calculate the cost of one trip to `cost.%of_trip%`*:*

```
apc_pac.map_cost_of_crr = “cost.of_trip”;
```

Exercise 11

~60mins

Exercise 11 summary



Chapter 12

Contexts

Iterators

Context

- All calculations have a context
- When Rave evaluates rules, the context is set to one trip, with only one leg in focus at any time
- It is the *application* that defines the context, not Rave
- When running reports, the context is *selected* by the planner
- Generating a report for all trips in the window puts all those trips into the context: `default_context`

Iterators

- Most calculations only move back and forth in a single chain. For example `next(leg(trip),…)`

- With iterators you may move across trips and rosters

- Mainly used for reports and scripts.

Iterators

- used with different traversers:
  - `count(iterator)`
  - `max(iterator, %duty_points%)`

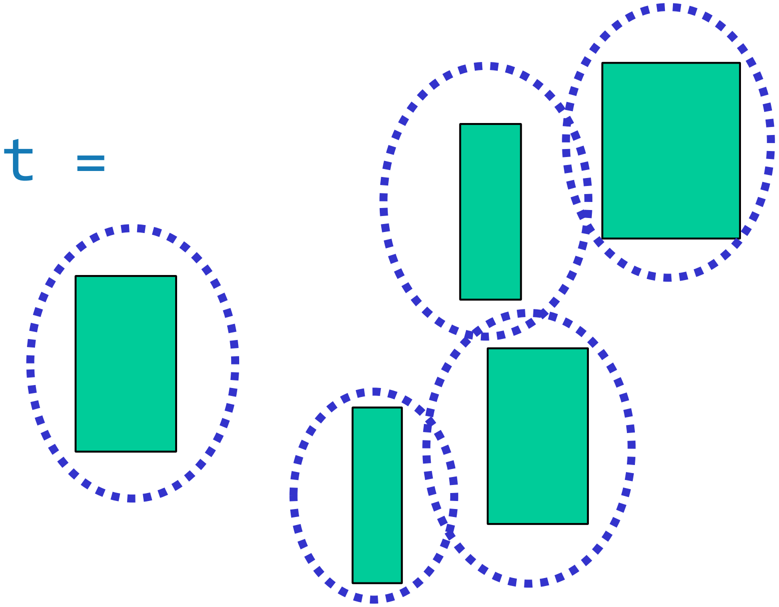
- the iterator generates a number of bags with objects
- the traverser evaluates one object per bag
- the simplest iterator is the one dividing all legs into separate bags, the `leg_set` iterator.

Iterators

leg_set

The `leg_set` iterator will put each leg in a separate group (bag) so that there is exactly one leg in each bag:

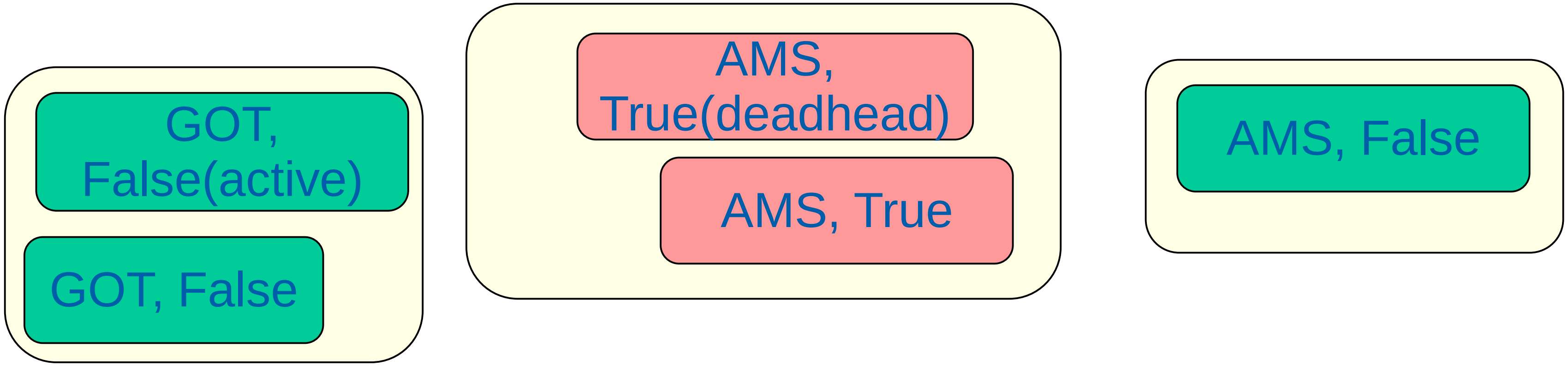
```
module iterators
global export iterator leg_set =
    partition (leg);
end
```



Example

Place legs with the same departure_airport_name and deadhead value in the same bag:

```
module iterators
global export iterator airport_set =
    partition (leg)
    by (departure_airport_name, deadhead);
end
```



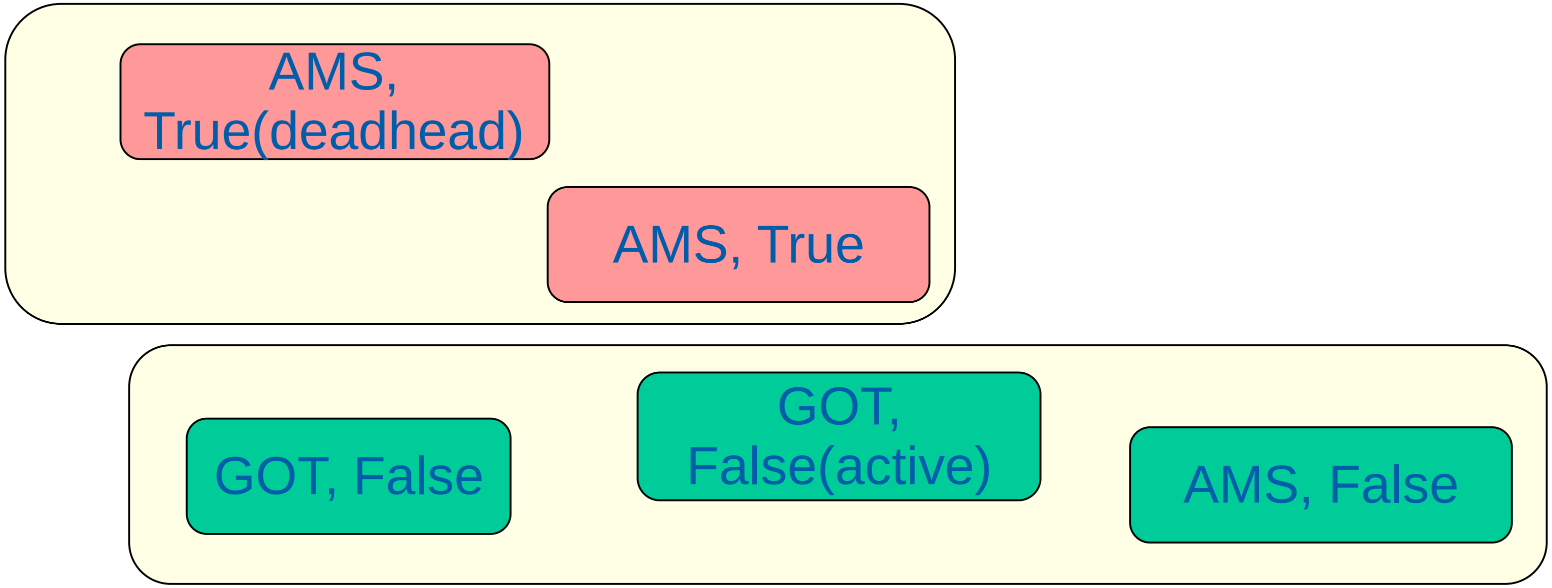
Iterators

Example

Place active and deadhead legs in different bags:

```
global export iterator active_set =
   partition (leg)
   by (deadhead);
end
```

There can only be 2 groups



AMS, True(deadhead)

AMS, True

GOT, False

GOT, False(active)

AMS, False

Iterators

Examples

Three examples using `leg_set`, `airport_set` and the five flights defined in the previous slides:

Sum block time of *all* flights

```
sum(leg_set, %block_time%)
```

Count the number of *active* departure stations

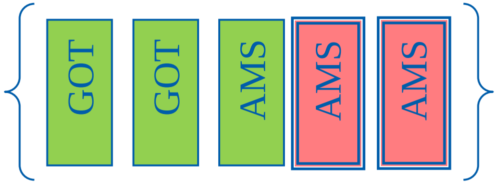
```
count (airport_set)
where (not deadhead)
```

Find max number of active departures from one station

```
max(airport_set, count(leg_set))
where (not deadhead)
```

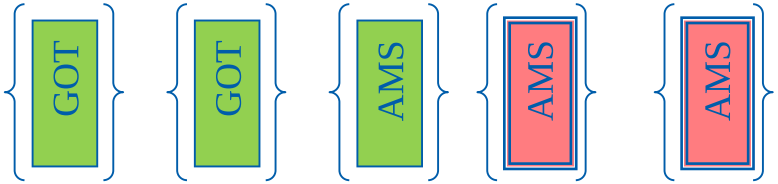
Iterators

Summarize block time



```
: sum(leg_set, %block_time%)          =

                          evaluate leg_set
```

sum(

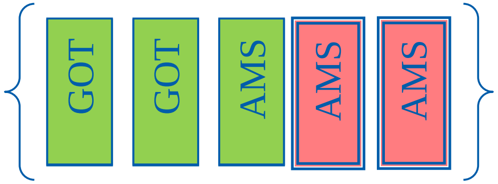


```
, %block_time% )          =

              evaluate %block_time%
```

```
sum (3:00 ; 1:30 ; 1:50 ; 1:00 ; 1:00 )          =
                                               8:20
```

Iterators

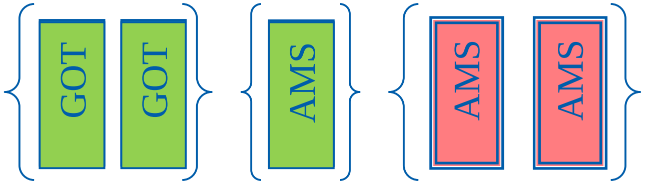
Count num active departure stations



:count(airport_set) where(not deadhead) =

evaluate airport_set

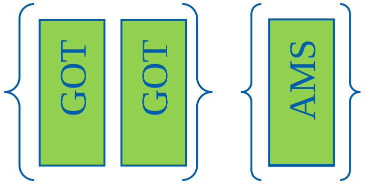
count(



) where(not deadhead) =

apply where

count(



)

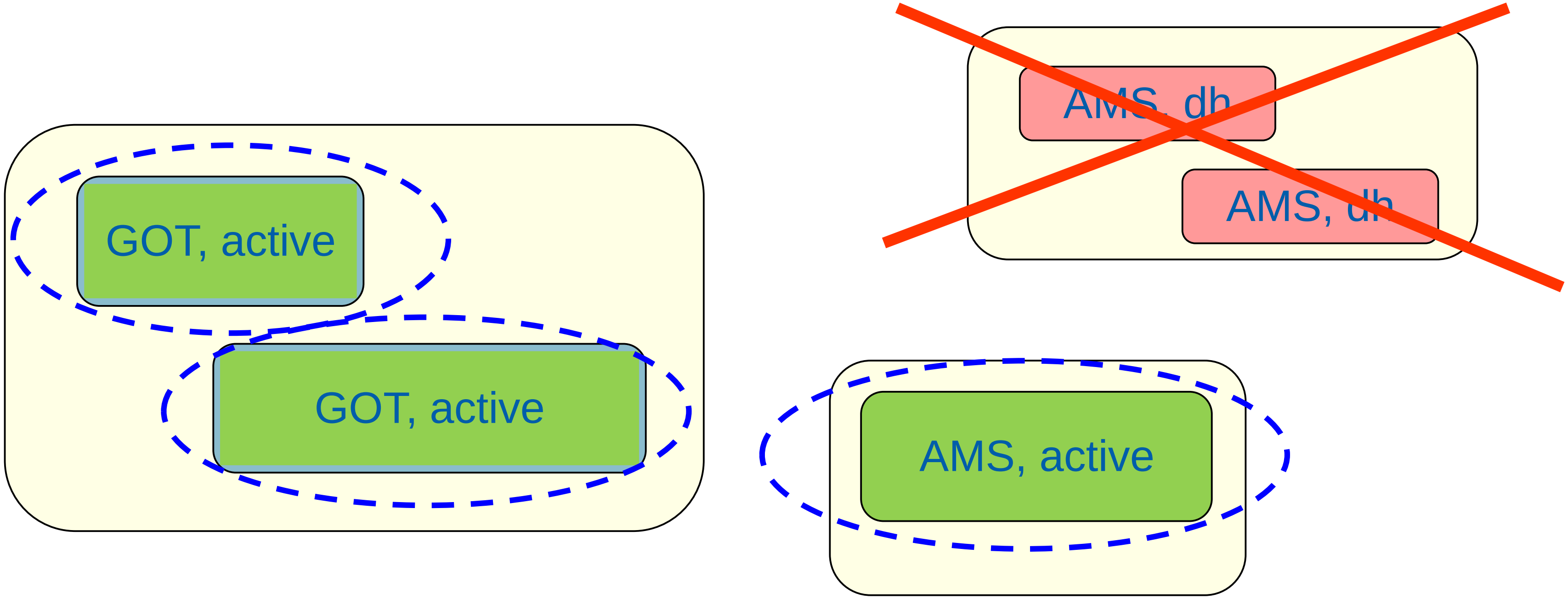
=

count 2

Iterators

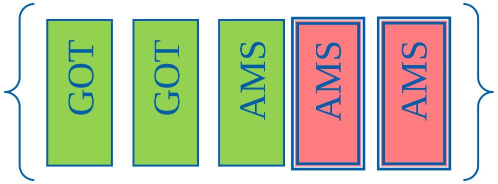
Max number of active departures from one station

For each `airport_set` bag, we put each leg in a separate `leg_set` bag, then count the number of `leg_set` bags.



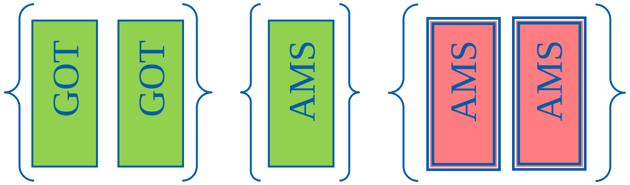
Iterators

Max number of active departures from one station



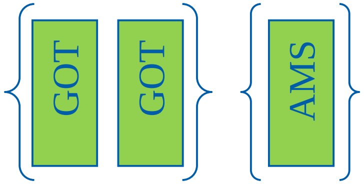
```
: max(airport_set, count(leg_set) )
  where (not deadhead)                  =
```

```
                          evaluate airport_set
```



```
max(  , count(leg_set) )
        where (not deadhead)            =
```

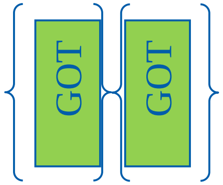
```
                          apply where
```



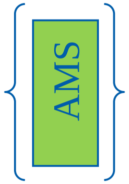
```
max(  , count(leg_set) )                =
```

```
                          apply count(leg_set)
```

```
max(
```



```
: 2
```



```
: 1    )                                =
                                        2
```

Final words

- Iterators should *only* be used in reports (and scripts)
- If used in Rave, always use atomic iterators (leg, duty, …) and explicit variables names:

  -
    ```
    %legs_in_bag% = count(leg_set);
    %duties_in_bag% = count(duty_set);
    ```

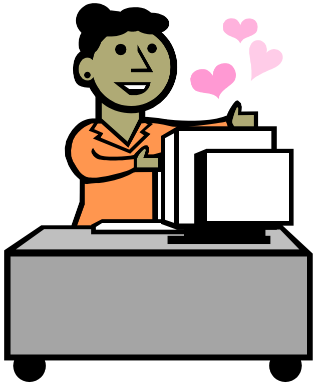
  compare with

  -
    ```
    %legs_in_trip% = count(leg(trip));
    ```
- Iterators will always work on the current context, ie they depend heavily on where they are called from.

Exercise 12

~120 mins

Exercise 12 summary



Chapter 13

Visualization

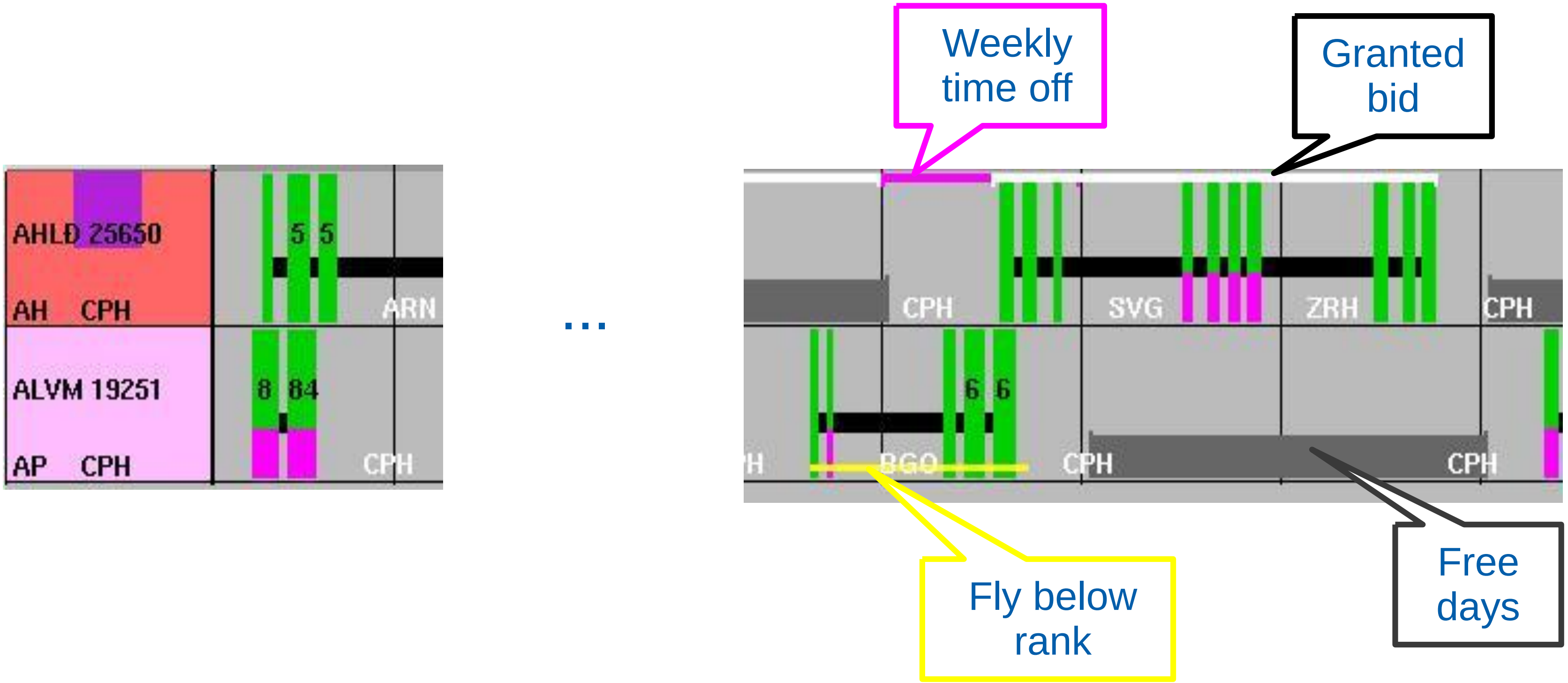
Rudobs

Sneak peak into GUI Customization

Visualisation

Header and leg colours

RUDOBS = RUle Defined OBjectS



Weekly time off

Granted bid

Fly below rank

Free days

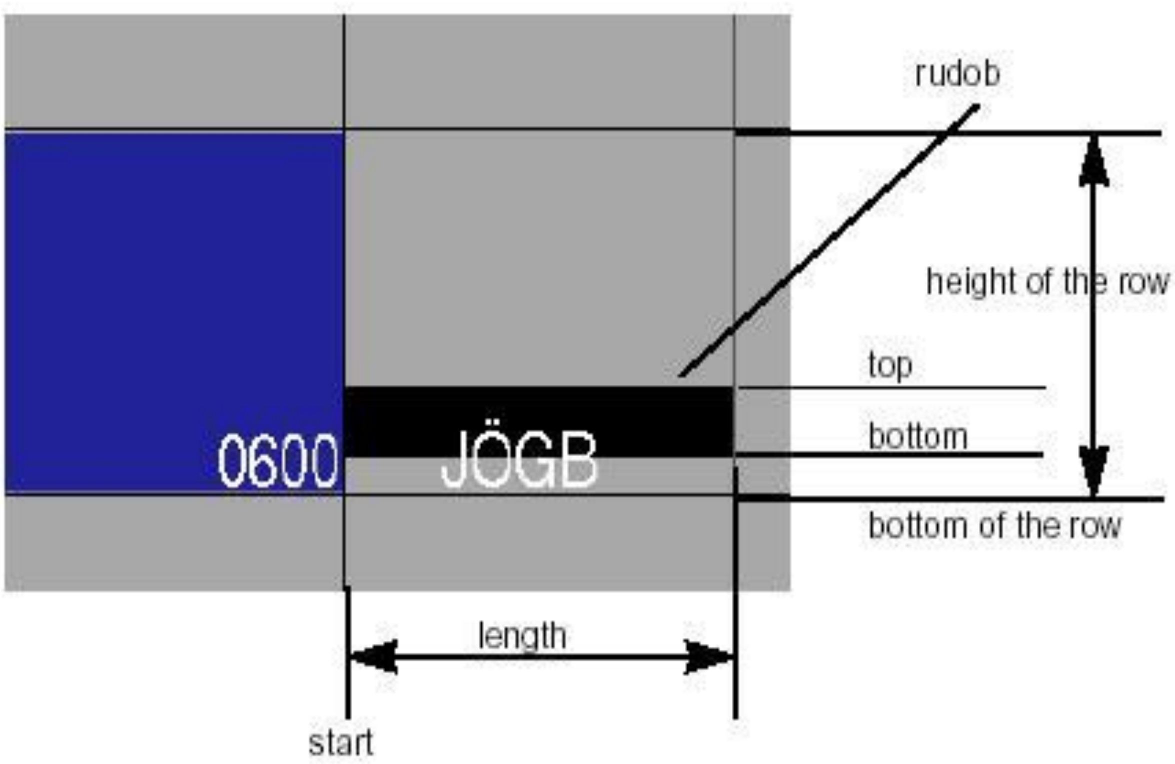
Visualisation

Used to:

- increase visibility
- highlight important information
- show rule values
- show warnings
- show illegality.

Covered in detail in the GUI course.

Rudobs


rudob
height of the row
top
bottom
0600
JÖGB
bottom of the row
length
start

Rudobs

map values

For each leg it is possible to define 100 Rudobs

To define Rudobs use map values:

```
map_rudob_len_[1..100]_[leg|rtd|crr|crew|ac|leg_set]
map_rudob_start_[1..100]_[leg|rtd|crr|crew|ac|leg_set]
map_rudob_color_[1..100]_[leg|rtd|crr|crew|ac|leg_set]
```

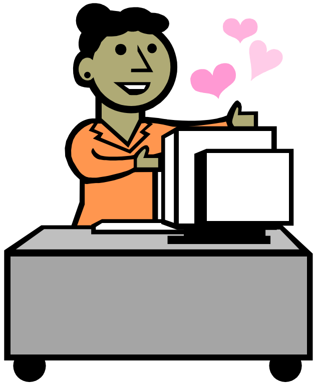
Example:

```
file report/studio_mappings
%map_rudob_text_1_crr% =
    “d_duty.rudob_rest_text”;
…
module d_duty
%rudob_rest_text% = …;
```

Exercise 13

~60 mins

Exercise 13 summary



Chapter 14

Final Words

Summary

Developing and testing

- Try not to disturb production
- Use hg or CVS
- Use your own test environment
- Comment your code
- Use remarks, also for Planners
- Indent your code
- Code standard – Best practice
- Use DWS
- Emacs mode, highlighting `/* -*-crc-*- */`

How 2 Think

- Make sure you understand the *details*
- Are there any *exceptions*?
- *Draw* an illegal chain on paper, try to find loopholes
- Actual value compared to limit value (param)
- Reuse (cost-) variables or build from scratch
- Valid statement, remark and planner remark
- *Test*, test and re-test; try to break your rule

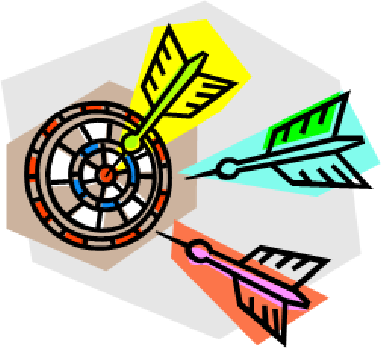
Course summary

You have learned:

- Rave syntax
- How to write Rave code and how it is used
- About variables, parameters, rules and costs
- How to find and use the on-line help and the Rave reference manual

You know how to:

- Update and maintain existing code
- Implement new functionality with Rave



Rave II

In Rave II you will:

- use modules
- write cost functions
- understand advanced Rave features
- understand how caching works
- avoid illegal sub-chain problems
- do performance analysis.

Other Rave courses

Rave Publisher I, II – Reports (PDL)
PRT (Python Report Toolkit)
Rave for Pairing Optimization
Rostering Optimization.

The end

This was Rave I

Welcome back to Jeppesen Crew Academy!