Jess Language: Controlling Execution

The Jess Language Part 3
CS3025, Knowledge-Based Systems
Lecture 10

Yuting Zhao
Yuting.zhao@gmail.com

2017-10-26

Outline

- **Multislots in templates (handling lists)**
- Language Control Structures
- Defining Functions
- Writing Interactive Programs
- Truth Maintenance
- Influence Order on the Agenda
- Modularisation

Multislots in Templates

- Templates can also have so-called **multislots**
  - These are slots that can hold lists

```
(deftemplate shopping-cart
        (slot      customer-id (type SYMBOL))
        (multislot basket))
```

- Use in pattern matching, with constraints:

Careful about Syntax !!

```
(defrule large-order-and-no-milk
   (shopping-cart
        (customer-id   ?id)
        (basket        $?cart & :(and (> (length$ ?cart) 50)
                                      (not (member$ milk ?cart))))
   =>
   (printout t "Do you need milk?" crlf))
```

Outline

- Multislots in templates (handling lists)
- **Language Control Structures**
- Defining Functions
- Writing Interactive Programs
- Truth Maintenance
- Influence Order on the Agenda
- Modularisation

Writing more complex RHS of Rules

- The RHS of a rule can consist of **a sequence of actions** that are executed during a Jess run:
  - assert, retract, modify
  - Functions, such as printout
- But also:
  - bind
  - if . . . then . . . Else
  - while
  - foreach
  - …

Note: whatever you can write at the RHS of a rule, you can directly type at the Jess command prompt

Control Statements on the RHS

IF – Then- Else

```
(defrule check-grocery-list
     (grocery-list $?x)
     =>
     (if (member$ eggs ?x)
      then
         (printout t "I need eggs" crlf)
      else
         (printout t "I don't need eggs" crlf))
)
```

- Can also be used at the command prompt:

```
Jess> (bind ?grocery-list (create$ eggs bread milk))
(eggs bread milk)
Jess> (if (member$ eggs ?grocery-list) then
(printout t "I need eggs" crlf)
else
(printout t "no eggs, thanks" crlf))
I need eggs
Jess>
```

Use Rules instead of if-statement

- **Option 1: using "test"**

```
(defrule check-grocery-list
    (grocery-list $?x)
    =>
    (if (member$ eggs ?x)
     then
        (printout t "I need eggs" crlf)
     else
        (printout t "I don't need eggs" crlf))
)
```

```
(defrule need-eggs
    (grocery-list $?grocery-list)
    (test (member$ eggs ?grocery-list))
    =>
    (printout t "I need eggs" crlf)
```

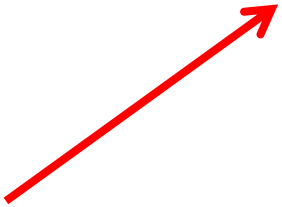
```
(defrule no-eggs
    (grocery-list $?grocery-list)
    (test (not(member$ eggs ?grocery-list))
    =>
    (printout t "No eggs" crlf)
```

Use Rules instead of if-statement

- **Option 2: using constraints**

```
(defrule need-eggs
     (grocery-list $?grocery-list&:(member$ eggs ?grocery-list))
     =>
     (printout t "I need eggs" crlf)
```

```
(defrule check-grocery-list
     (grocery-list $?x)
     =>
     (if (member$ eggs ?x)
      then
         (printout t "I need eggs" crlf)
      else
         (printout t "I don't need eggs" crlf))
)
```



```
(defrule no-eggs
     (grocery-list $?grocery-list&:(not(member$ eggs ?grocery-list)))
     =>
     (printout t "No eggs" crlf)
```

Control Statements on the RHS

For-Each, While

- **for each**:

```
(defrule check-grocery-list
    (grocery-list $?grocery-list)
    =>
    (bind ?i 1)
    (foreach ?e ?grocery-list
        (printout t "Item " ?i ": " ?e crlf)
        (bind ?i (+ ?i 1))
    )
)
```

- **while**:

```
(defrule check-grocery-list
    (grocery-list $?grocery-list)
    =>
    (bind ?i 1)
    (while (<= ?i (length$ ?grocery-list)) do
        (bind ?e (nth$ ?i ?grocery-list))
        (printout t "Item " ?i ": " ?e crlf)
        (bind ?i (+ ?i 1))
    )
)
```

- How to assign value to a variable
- operators: + - * /
- brackets

Outline

- Multislots in templates (handling lists)
- Language Control Structures
- **Defining Functions**
- Writing Interactive Programs
- Truth Maintenance
- Influence Order on the Agenda
- Modularisation

Defining Functions

- Jess provides the construct "**deffunction**":

```
(deffunction min (?a ?b)
     (if (< ?a ?b) then
         (return ?a)
      else
         (return ?b))
)

(deffunction min ($?args)
     (bind ?minval (nth$ 1 ?args))
     (foreach ?n ?args
         (if (< ?n ?minval) then (bind ?minval ?n))
     )
     ;(return ?minval)
     ?minval
)
```

```
(deffunction min (?a ?b)
     (if (< ?a ?b) then
         ?a
      else
         ?b)
)
```

- argument
- return

The value of the last element in a function is returned:
We can use "return" or directly add the variable as the final statement

Use deffunction in Rules

- On the LHS in constraints:

```
(deffunction is-positive (?v)
        (>= ?v 0)
)
```

```
(deffunction is-negative (?v)
        (< ?v 0)
)
```

```
(defrule check-value-positive
    (value ?value&:(is-positive ?value))
    =>
    (printout t "Value is positive" crlf))
```

```
(defrule check-value-negative
    (value ?value&:(is-negative ?value))
    =>
    (printout t "Value is negative" crlf))
```

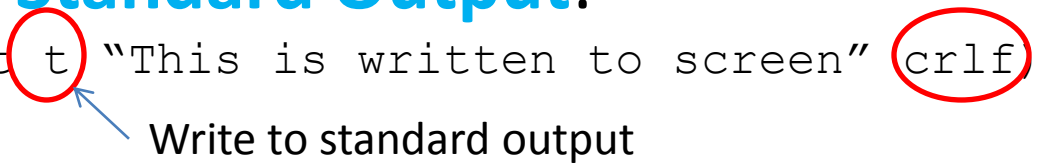
- load (call)
- argument
- return

Outline

- Multislots in templates (handling lists)
- Language Control Structures
- Defining Functions
- **Writing Interactive Programs**
- Truth Maintenance
- Influence Order on the Agenda
- Modularisation

Writing Interactive Programs

- Writing to **Standard Output**:
  - `(printout t "This is written to screen" crlf)`



- We can also read from **Standard Input** (your keyboard), using:
  - **`(read), (readline),`**
  - This will stop Jess executing and wait for the user to provide some input
  - The input read from standard input can be explicitly bound to a variable:
    - **`(bind ?answer (read))`**

Simple text-based User Input

- Read user input from the command line at the **RHS of a rule**:

```
(assert (ask-question "blabla"))
(defrule ask-user
     (ask-question ?question)
     =>
     (printout t ?question " ")
     (bind ?answer (read))
     (assert (answer ?answer)))
```

Read user input

- Can be used in other **functions**:

```
(deffunction ask-user (?question)
     (printout t ?question " ")
     (return (read)))
```

Read user input

Outline

- Multislots in templates (handling lists)
- Language Control Structures
- Defining Functions
- Writing Interactive Programs
- **Truth Maintenance**
- Influence Order on the Agenda
- Modularisation

Dependencies between Facts

- We create rules to simulate a simple light switch:
  - We want to change the situation with only one fact to be asserted or retracted (simulate the light being switched on / off)
- Simpler: use "logical"

```
(defrule light-switch-on
    (switch on)
    (not (light shines))
    =>
    (assert (light shines)))
(defrule light-switch-off
    ?f <- (light shines)
    (not (switch on))
    =>
    (retract ?f))
```

```
(reset)
(bind ?x (assert (switch on)))
(agenda)
(retract ?x)
(agenda)
```

Conditional Element - logical

- With the “logical” conditional element, a ***logical dependency*** between facts can be established:

```
(defrule light-shines-while-switch-on
    (logical (switch on))
    =>
    (assert (light shines)))
```

- If the fact `(switch on)` is asserted, this rule will fire and assert the fact `(light shines)`
- If the fact `(switch on)` is retracted, the connected fact `(light shines)` is retracted as well

```
(reset)
(bind ?x (assert (switch on)))
(agenda)
(retract ?x)
(agenda)
```

Conditional Element - logical

```
(defrule gadgets-on-while-switch-on
        (logical (switch on))
        =>
        (assert (is-on television))
        (assert (is-on video))
)
TRUE
Jess> (assert (switch on))
<Fact-0>
Jess> (run)
1
Jess> (facts)
f-0    (MAIN::switch on)
f-1    (MAIN::is-on television)
f-2    (MAIN::is-on video)
For a total of 3 facts in module MAIN.
Jess> (retract 0)
TRUE
Jess> (facts)
For a total of 0 facts in module MAIN.
Jess>
```

Create logical connection between this fact and any fact asserted with this rule

Assert fact

Execute RHS of rule

All three facts in WM

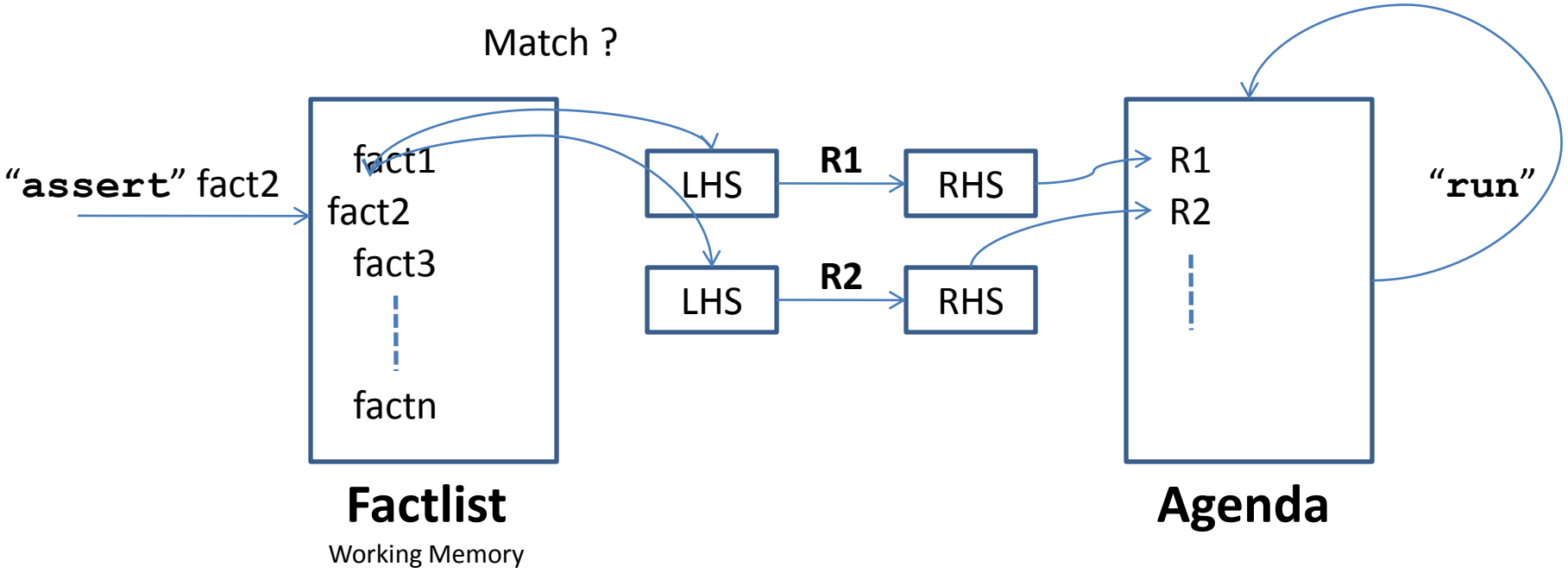
Retract fact

All facts removed

Outline

- Multislots in templates (handling lists)
- Language Control Structures
- Defining Functions
- Writing Interactive Programs
- Truth Maintenance
- **Influence Order on the Agenda**
  - Salience,
  - Conflict Resolution
- Modularisation

Jess Execution Revisited



- How can we **influence the ordering of rule activations** on the Agenda?

Salience

- We can define a "**salience**" for each rule
  - Is a number between -10000 and +10000
  - The default salience of a rule is 0
  - Introduces a ranking between rules:
    - A rule with higher salience will always fire before a rule with lower salience
    - Salience influences the ordering of rules on the Agenda
- Syntax:
  - (declare (salience <some-number>))

```
(defrule my-rule
      (declare (salience 10))
      . . . )
```

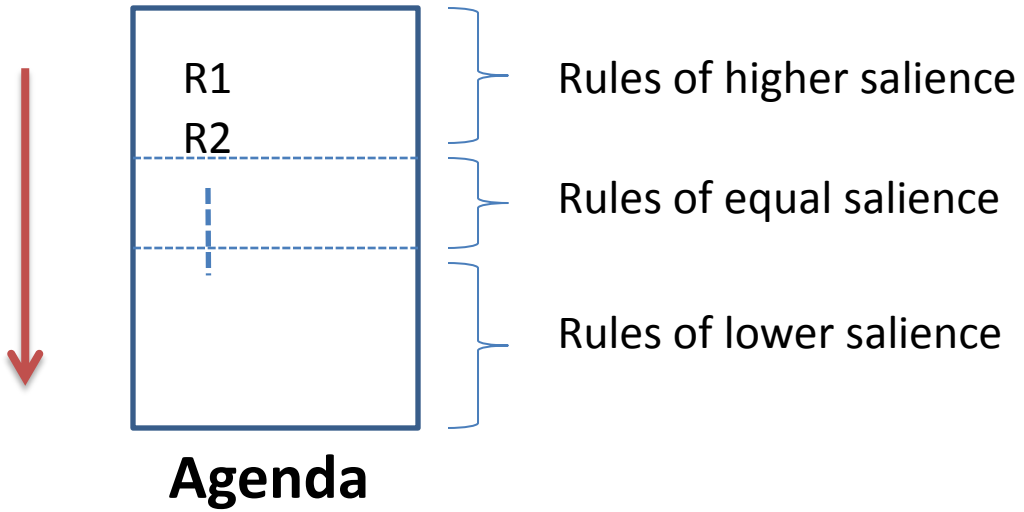
Salience Evaluation Method

(set-salience-evaluation (*when-defined* | *when-activated* | *every-cycle*))

- Salience evaluation methods:
  - When **defined**: this is the default behaviour, the salience is fixed at design time
  - When **activated**: when the rule is activated, the salience is evaluated and determines how it is written on the agenda
  - **Every cycle**: after each firing of a top rule on the agenda, the salience of all remaining activated rules on the agenda is re-evaluated, which results in a re-ordering of rules on the agenda (is computationally expensive)
- Function to query what method is currently set:
  - `(get-salience-evaluation)`

Salience and the Agenda

- Rules of equal salience have equal precedence
- A rule with higher salience is always higher on the agenda and will always fire first



Conflict Resolution Strategies (**CRS**)

- In general:
  - If two rules on the agenda have the **same salience** (which is mostly the case), which one fires first?
- Jess uses a *conflict resolution strategy (CRS)* to determine their order on the agenda
  - Jess has two built-in strategies that can be used
    - **Depth** (Last In First Out)
    - **Breadth** (First In First Out)
  - The default CRS is the ***depth*** strategy

The Depth Conflict Resolution Strategy

(Default Strategy)

- *A newly activated rule is placed **above** all others of the same salience (LIFO "Last In First Out")*

- E.g.: Rule r2 is activated with fact f-5:

10	r1: f-1, f-3
10	r1: f-1, f-2
0	r4: f-2, f-4
0	r2: f-1
-10	r3: f-4

10	r1: f-1, f-3
10	r1: f-1, f-2
0	r2: f-5
0	r4: f-2, f-4
0	r2: f-1
-10	r3: f-4

State of the Agenda before / after Rule activation

The **Breadth** Conflict Resolution Strategy

- *A newly activated rule is placed **below** all others of the same salience (FIFO "First In First Out")*

- E.g.: Rule r2 is activated with fact f-5:

10	r1: f-1, f-3
10	r1: f-1, f-2
0	r4: f-2, f-4
0	r2: f-1
-10	r3: f-4

10	r1: f-1, f-3
10	r1: f-1, f-2
0	r4: f-2, f-4
0	r2: f-1
0	r2: f-5
-10	r3: f-4

State of the Agenda before / after Rule activation

Conflict Resolution Strategy – Depth
(Default Strategy)

- Can be set with the following command:
  - **(set-strategy depth)**
    - Makes rules fire in reversed order of activation, the most recently activated rule fires first, *default strategy*

```
(deffacts item-list
     (item 1)
     (item 2)
     (item 3)
     (item 4)
     (item 5)
     (item 6)
     (item 7)
     (item 8)
     (item 9)
     (item 10)
)

(set-strategy depth)
(defrule produce-list
     (initial-fact)
     (item ?c)
     =>
     (printout t "Item: " ?c crlf)
)
```

```
[Activation: MAIN::produce-list  f-0, f-10 ; time=11 ; totalTime=12 ; salience=0]
[Activation: MAIN::produce-list  f-0, f-9 ; time=10 ; totalTime=11 ; salience=0]
[Activation: MAIN::produce-list  f-0, f-8 ; time=9 ; totalTime=10 ; salience=0]
[Activation: MAIN::produce-list  f-0, f-7 ; time=8 ; totalTime=9 ; salience=0]
[Activation: MAIN::produce-list  f-0, f-6 ; time=7 ; totalTime=8 ; salience=0]
[Activation: MAIN::produce-list  f-0, f-5 ; time=6 ; totalTime=7 ; salience=0]
[Activation: MAIN::produce-list  f-0, f-4 ; time=5 ; totalTime=6 ; salience=0]
[Activation: MAIN::produce-list  f-0, f-3 ; time=4 ; totalTime=5 ; salience=0]
[Activation: MAIN::produce-list  f-0, f-2 ; time=3 ; totalTime=4 ; salience=0]
[Activation: MAIN::produce-list  f-0, f-1 ; time=2 ; totalTime=3 ; salience=0]
For a total of 10 activations in module MAIN.
Item: 10
Item: 9
Item: 8
Item: 7
Item: 6
Item: 5
Item: 4
Item: 3
Item: 2
Item: 1
```

Last In First Out
(item 10)

Conflict Resolution Strategy - Breadth

- Can be set with the following command:
  - **(set-strategy breadth)**
    - Makes rules fire in order of activation, the most recently activated rule fires last

```
(deffacts item-list
    (item 1)
    (item 2)
    (item 3)
    (item 4)
    (item 5)
    (item 6)
    (item 7)
    (item 8)
    (item 9)
    (item 10)
)

(set-strategy breadth)
(defrule produce-list
    (initial-fact)
    (item ?c)
    =>
    (printout t "Item: " ?c crlf)
)
```

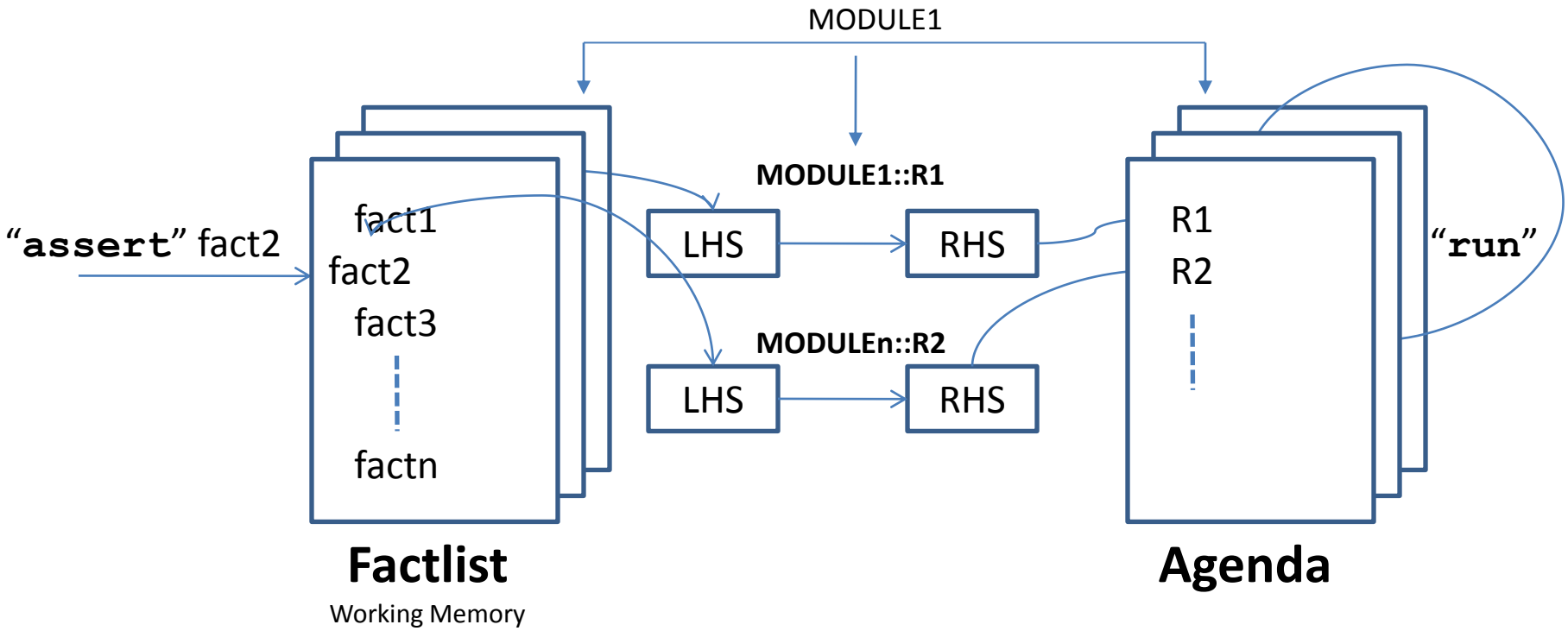
```
[Activation: MAIN::produce-list  f-0, f-1 ; time=2 ; totalTime=3 ; salience=0]
[Activation: MAIN::produce-list  f-0, f-2 ; time=3 ; totalTime=4 ; salience=0]
[Activation: MAIN::produce-list  f-0, f-3 ; time=4 ; totalTime=5 ; salience=0]
[Activation: MAIN::produce-list  f-0, f-4 ; time=5 ; totalTime=6 ; salience=0]
[Activation: MAIN::produce-list  f-0, f-5 ; time=6 ; totalTime=7 ; salience=0]
[Activation: MAIN::produce-list  f-0, f-6 ; time=7 ; totalTime=8 ; salience=0]
[Activation: MAIN::produce-list  f-0, f-7 ; time=8 ; totalTime=9 ; salience=0]
[Activation: MAIN::produce-list  f-0, f-8 ; time=9 ; totalTime=10 ; salience=0]
[Activation: MAIN::produce-list  f-0, f-9 ; time=10 ; totalTime=11 ; salience=0]
[Activation: MAIN::produce-list  f-0, f-10 ; time=11 ; totalTime=12 ; salience=0]
For a total of 10 activations in module MAIN.
Item: 1
Item: 2
Item: 3
Item: 4
Item: 5
Item: 6
Item: 7
Item: 8
Item: 9
Item: 10
```

Outline

- Multislots in templates (handling lists)
- Language Control Structures
- Defining Functions
- Writing Interactive Programs
- Truth Maintenance
- Influence Order on the Agenda
- **Modularisation**

Modules

- Jess supports a modular design of programs – a Jess program can be separated into modules
- A module is a named subset of the **rules**, **deftemplates**, **ordered facts** and other Jess **constructs**
- This is useful, if you want to separate pieces of code into sections, where each module solves a separate problem



Modules

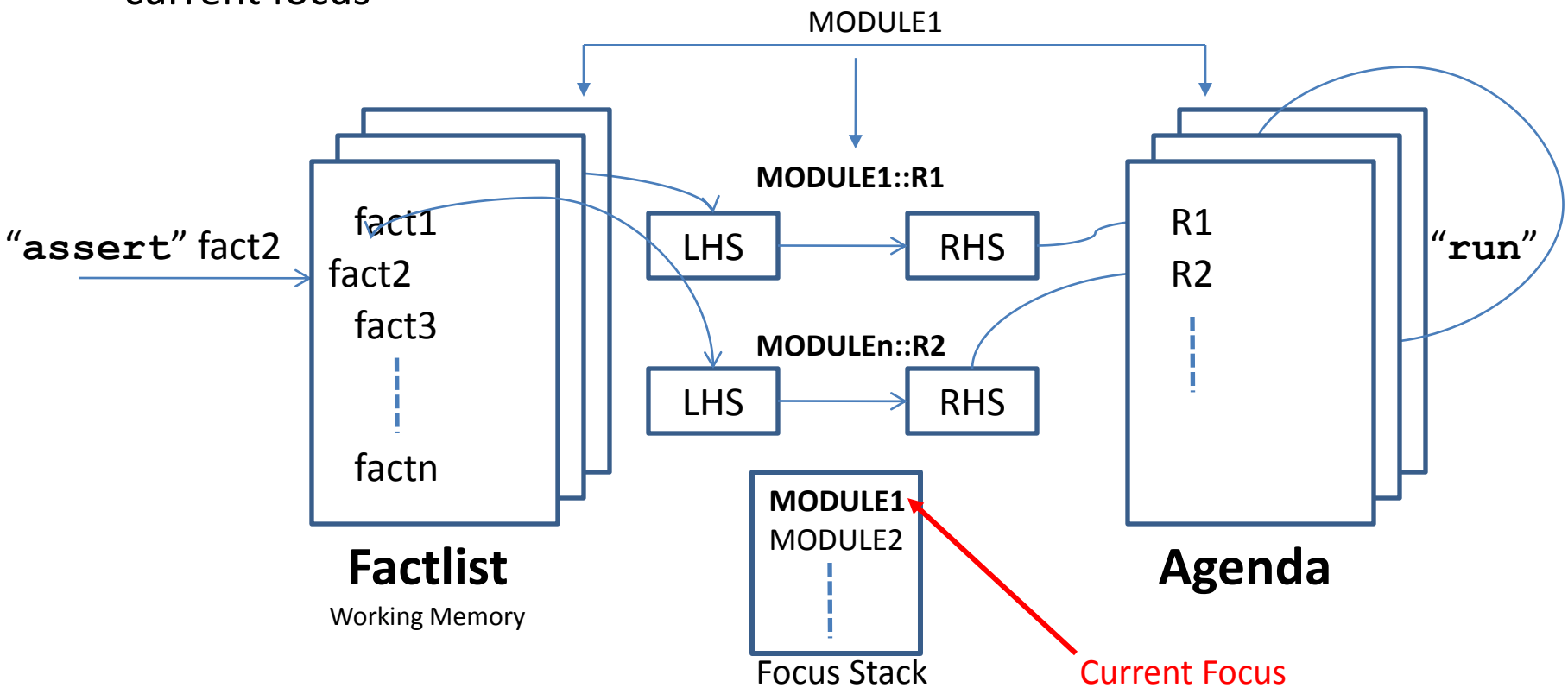
- A module defines a **namespace** for Jess constructs such as templates and rules
  - We can regard each module having its own agenda and fact list: each fact and each rule has a module prefix
  - We know already the default module **MAIN**

```
Jess> (assert (switch on))
<Fact-0>
Jess> (facts)
f-0    (MAIN::switch on)
```

Fact (switch on) is in default module **MAIN**

Modules – Execution

- For execution, Jess maintains a module "focus"
  - A particular module is the "focus" of execution
  - Default focus:
    - Is determined by the most recent "defmodule" statement (if there is no defmodule, it is MAIN)
- Jess maintains a "focus stack", with the top element representing the current focus



Module - Execution

- Default focus:
  - Is determined by the **most recent** `defmodule` **statement** (if there is no `defmodule`, it is MAIN)
- Specifying a focus explicitly during execution (on the RHS of a rule):
  - `(focus <module name>)`
- Auto focus:
  - When a rule declares the `autofocus` property, its own module will get the focus automatically
- During execution, I can access facts from other modules by explicitly qualifying them with a module name

Example

```
(clear)
(defmodule HOME)
(deftemplate hobby (slot name)(slot income))

(defmodule WORK)
(deftemplate job (slot salary))

(defrule WORK::quit-job
    ;(declare (auto-focus TRUE))
    (job (salary ?s))
    (HOME::hobby (income ?i & :(> ?i (/ ?s 2))))
    (mortgage-payment ?m & :(< ?m ?i))
    =>
    (printout t "Call your boss and quit your job!" crlf)
)


(reset)
(focus HOME)
(assert (mortgage-payment 2000))
(assert (job (salary 2000)))
(assert (HOME::hobby (income 4000)))
(facts *)
(agenda *)
(run)
```

Define module HOME

Define module WORK

If auto-focus is activated, the focus will switch to WORK automatically

The fact “hobby” is owned by module HOME

With “focus”, the module HOME is put into focus (pushed on the focus stack)
We will get an activation of rule WORK::quit-job, but no execution, as the focus is on module HOME

Summary

- Controlling Execution
  - …
- Question?