Rules in production system

Production systems formalize knowledge in a certain form called production rules and use forward-chaining reasoning to derive new things. They are used in many practical applications (e.g., expert systems).

A production system maintains a working memory (WM) which contains assertions that are changing during the operation of the system.

A **production rule** consists of an antecedent set of conditions and a consequent set of actions:

IF conditions THEN actions

The antecedent conditions are tests applied to the current state of the WM; the consequent actions are a set of actions that modify the WM.

Rules in production system

The production system operates in a three-step cycle that repeats until no more rules are applicable to the WM:

1. Recognize – find the applicable rules, i.e. those rules whose antecedent conditions are satisfied by the current WM
2. Resolve conflicts – among the rules found at the first step (called conflict set), choose those to fire
3. Act – change the WM by performing the consequent actions of all the rules selected at step 2

Rules in production system

Working Memory

WM consists of a set of working memory elements (WMEs).

A WME has the from:

(type attribute$_1$:value$_1$…attribute$_n$:value$_n$)

where type, attribute$_i$ and value$_i$ are atoms.

Examples

(person age:21 home:bucharest)

(student name:john dept:computerScience)

Rules in production system

Declaratively, a WME is an existential sentence:

$$\exists x[type(x) \wedge attribute_1(x)=value_1 \wedge \ldots \wedge attribute_n(x)=value_n]$$

WMEs represent objects and relationships between them can be handled by reification (i.e., transforming a sentence into an object)

Purchases(john,bike,nov11)

Purchases(p3) ^ agent(p3)=john ^ object(p3)=bike…

For example,

(basicFact relation:olderThan firstArg:john secondArg:mary)

may be used to express the fact that John is older than Mary.

Rules in production system

Production rules

The conditions of a rule are connected by conjunctions. A condition can be positive or negative (written as —cond) and has the form:

$$(\text{type attribute}_1\text{:specification}_1 \ldots \text{attribute}_k\text{: specification}_k)$$

where each specification is one of the following:

- an atom
- a variable
- an evaluable expression within []
- a test within {}
- conjunction, disjunction or negation of a specification

Rules in production system

For instance,

(person age:[n+1] height:x)

is satisfied if there is a WME whose type is person and whose age attribute is n+1, where n is specified elsewhere.

If the variable x is not bound, then x will be bound with the value of the attribute height; otherwise, the value of the attribute height in the WME has to be the same as the value of x.

—(person age:{<20 ∧ >6}) is satisfied if there is no WME in WM whose type is person and age is between 6 and 20 (negation as failure)

Rules in production system

A WME matches a condition if the types are identical and for each attribute/specification pair in the condition, there is a corresponding attribute/value pair in the WME and the value matches the specification.

The matching WME may have other attributes that are not mentioned in the condition:

condition (control phase:5)

matching WME (control phase:5 position:7…)

Rules in production system

The consequent set of actions of production rules have a procedural interpretation. All actions are executed in sequence and they can be of the following types:

1. ADD – adds a new WME to WM
2. REMOVE *i* – removes from WM the WME that matches the i^{th} condition in the antecedent of the rule; not applicable if the condition is negative
3. MODIFY *i* (attribute specification) – modifies the WME that matches the i^{th} condition in the antecedent by replacing its current value for attribute by specification; not applicable if the condition is negative.

Obs. In ADD and MODIFY, the variables that appear refer to the values obtained when matching the antecedent of the rule.

Rules in production system

Example 1

IF (student name:x) THEN ADD (person name:x)

(the equivalent of ∀x.Student(x) ⊃ Person(x))

Example 2 – assume that a rule added a WME of type birthday

IF (person age:x name:n) (birthday who:n)

THEN MODIFY 1 (age [x+1])

REMOVE 2

Rules in production system

Example 1

IF (student name:x) THEN ADD (person name:x)

(the equivalent of ∀x.Student(x) ⊃ Person(x))

Example 2 – assume that a rule added a WME of type birthday

IF (person age:x name:n) (birthday who:n)

THEN MODIFY 1 (age [x+1])

REMOVE 2

/

to prevent the rule from firing again

Rules in production system

Example 3 – to indicate the phase of computation

IF (starting) THEN REMOVE 1

ADD (control phase:1)

…

IF (control phase:x) …other conditions

THEN MODIFY 1 (phase [x+1])

IF (control phase:5) THEN REMOVE 1

Rules in production system

Example 4 – we have three cubes in a heap, each of different size. With a robotic hand, we want to move the cubes in the positions named 1, 2 and 3. The goal is to place the largest cube in position 1, the middle one in 2 and the smallest one in position 3.

WM

1. (counter value:1)
2. (cube name:A size:10 position:heap)
3. (cube name:B size:30 position:heap)
4. (cube name:C size:20 position:heap)

Rules

1. IF (cube position:heap name:n size:s)
   —(cube position:heap size:{>s})
   —(cube position:hand)
   THEN MODIFY 1 (position hand)
2. IF (cube position:hand) (counter value:i)
   THEN MODIFY 1 (position i)
   MODIFY 2 (value [i+1])

there are no conflicts because only one rule can fire at a time

Rules in production system

System operation:

Rule 1 → n=B, s=30 →WME3 changes to (cube name:B size:30 position:hand)

Rule 2 → i=1 → WME3 changes to (cube name:B size:30 position:1)
WME1 changes to (counter value:2)

Rule 1 → n=C, s=20 →WME4 changes to (cube name:C size:20 position:hand)

Rule 2 → i=2 → WME4 changes to (cube name:C size:20 position:2)
WME1 changes to (counter value:3)

Rule 1 → n=A, s=10 →WME2 changes to (cube name:A size:10 position:hand)

Rule 2 → i=3 → WME2 changes to (cube name:A size:10 position:3)
WME1 changes to (counter value:4)

No rule is applicable now, so the production system halts.

Final WM
1. (counter value:4)
2. (cube name :A size:10 position:3)
3. (cube name :B size:30 position:1)
4. (cube name :C size:20 position:2)

Rules in production system

Example 5
Compute the greatest common factor of two integers.
Represent the initial WM, the rules of the production system and the final WM after the system operates.

Example: represent the input values 6 and 9 in the initial WM; after the system runs, the value 3 should be in the final WM.

Rules in production system

gcf(0,a)=gcf(a,0)=a

if a>b then gcf(a,b)=gcf(a-b,b)

else gcf(a,b)=gcf(a,b-a)

initial WM (gcf val1:6 val2:9)

1. IF (gcf val1:x val2:y) THEN ADD(prod val:[x*y] val1:x val2:y)
   REMOVE 1
2. IF (prod val:0 val1:x val2:y) THEN ADD(res val:[x+y])
   REMOVE 1
3. IF(prod val:{≠0} val1:x val2:y) (prod val1:{>y}) THEN MODIFY 1 (val1 [x-y])
   MODIFY 1 (val [(x-y)*y])
4. IF(prod val:{≠0} val1:x val2:y) (prod val1:{≤y}) THEN MODIFY 1 (val2 [y-x])
   MODIFY 1 (val [x*(y-x)])

final WM (res val:3)

All conditions are disjoint, one rule fires at a time.

Rules in production system – solving conflicts

There are conflict resolution strategies to eliminate some applicable rules, if necessary.

The most common approaches are:

- **Order:** choose the first applicable rule in order of presentation (this strategy is implemented in PROLOG)
- **Specificity:** choose the applicable rule whose conditions are most specific

IF (bird) THEN ADD (canFly)

More specific than the first:

- IF (bird weight:{>100}) THEN ADD (cannotFly)
- IF (bird)(penguin) THEN ADD (cannotFly)

Another criterion should be applied further to choose between the two rules

- **Recency:** choose the applicable rule that matches the most/least recently created/modified WME
- **Refractoriness:** reject a rule that has just been applied with the same value of its variables – it prevents going into loops by repeated firing of a rule because of the same WME.

Rules in production system – efficiency

The rule matching operation consumes up to 90% of the operating time of a production system.

Two key observations led to a very efficient implementation:

- WM modifies very little during an execution cycle
- Many rules share common conditions

The **RETE** algorithm [1] creates a network from the rules' antecedents. This network can be created in advance, as the rules do not change during the system operation.

During operation, new or changed WMEs are checked if they satisfy the conditions of a rule. In this way, only a small part of WM is re-matched against the conditions of the rules, reducing drastically the time to calculate the applicable rules.

CLIPS (C Language Integrated Production System) based on RETE – for building expert systems.

[1] Charles Forgy: *Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem.* In: Artificial Intelligence, vol. 19, pp. 17–37, 1982.

Rules in production system – advantages

- Modularity – because the rules work independently, they can be added or deleted relatively easily – but for large system, this advantage fades away
- Control – a (relatively) simple control structure
- Transparency – the rules use a terminology easy to understand and the reasoning can be traced and explained in natural language

Production systems are used in a wide variety of practical problems (e.g., medical diagnosis, checking for eligibility for credits, configuration of systems).

Rules in production system – applications

MYCIN – expert system for diagnosis of bacterial infections, developed at Stanford University in ‘70

– 500 production rules for recognizing 100 causes of infections

– the most significant contribution was the introduction of a level of certainty of evidences and confidence in hypothesis

XCON – rule-based system for configuring computers, developed at Carnegie Mellon University in 1978

– 10000 rules to describe hundreds of types of components

– contributed to a growing commercial interest in rule-based expert systems.

For an expert system example based on MYCIN uncertainty factors in Prolog please see:

1. Florentina Hristea, Maria Florina Balcan. Căutarea şi reprezentarea cunoştinţelor în Inteligenţa artificială. Teorie şi aplicaţii, Editura Universităţii din Bucureşti, 2005 – chapter 4.3.6

Grammars in Prolog - the DCG notation (cont…)

Input file with ‘sentences’ of different types, written as lists:

[**rule**, 1, **if**, budget_low, **and**, summer, **then**, at_costinesti].

[**goal**, **is**, at_costinesti].

[**ask**, season, **options**, **‘(‘**, summer, winter, **‘)’**, **message**, ’what season do you prefer?’].

Grammars in Prolog - the DCG notation

```
:-dynamic goal/1.
:-dynamic rule/3.
:-dynamic question/3.

process_sentence:-read(S), translate(R,S,[]), assertz(R).

translate(goal(X)) --> [goal, is, X].

translate(rule(N,premises(If),conclusion(Then))) -->identificator(N), if(If), then(Then).

translate(question(Atr,M,Message)) -->[ask,Atr], lista_optiuni(M), afiseaza(Message).
```

Grammars in Prolog - the DCG notation

```
identificator(N) -->[rule, N].

if(If) -->[if], lista_premise(If).

lista_premise([n(If)])--> [If], [then].
lista_premise([n(First)|Rest]) --> [First], [and], lista_premise(Rest).

then(Then) -->[Then].
```

Grammars in Prolog - the DCG notation

```
lista_optiuni(M) -->[options,'('], lista_de_optiuni(M).

lista_de_optiuni([Element]) -->[Element,')'].
lista_de_optiuni([Element|T]) -->[Element], lista_de_optiuni(T).

afiseaza(Message) -->[message,Message].
```

Grammars in Prolog - the DCG notation

%S=[rule,1, if, budget_low, and, summer, then, at_costinesti]

?- process_sentence, rule(X,Y,Z).

X = 1,

Y = premises([n(budget_low), n(summer)]),

Z = conclusion(at_costinesti)

%S=[ask, season, options, '(', summer, winter, ')', message, 'what season do you prefer?']

?- process_sentence, question(A,B,C).

A = season,

B = [summer, winter],

C = 'what season do you prefer?'.

Defining operators in Prolog

The expression 2*3+3*4 can be written in Prolog in the equivalent form +(*(2,3), *(3,4)).

+ and * are predefined operators and + has a higher precedence than *.

In Prolog, we can define operators other than the predefined ones. There are 3 types of operators:

Infix operators: xfx, xfy, yfx

Prefix operators: fx, fy

Postfix operators: xf, yf

'f' represents the operator and 'x' and 'y' represent the arguments. To explain the difference between 'x' and 'y' we need to introduce the precedence of an argument. If an argument appears between parentheses or if it is a simple object then the precedence is 0; if the argument is a structure, then its precedence is the precedence of the main functor.

'x' represents an argument whose precedence must be strictly less than the precedence of the operator 'f' and 'y' represents an argument whose precedence is less than or equal to the precedence of the 'f' operator.

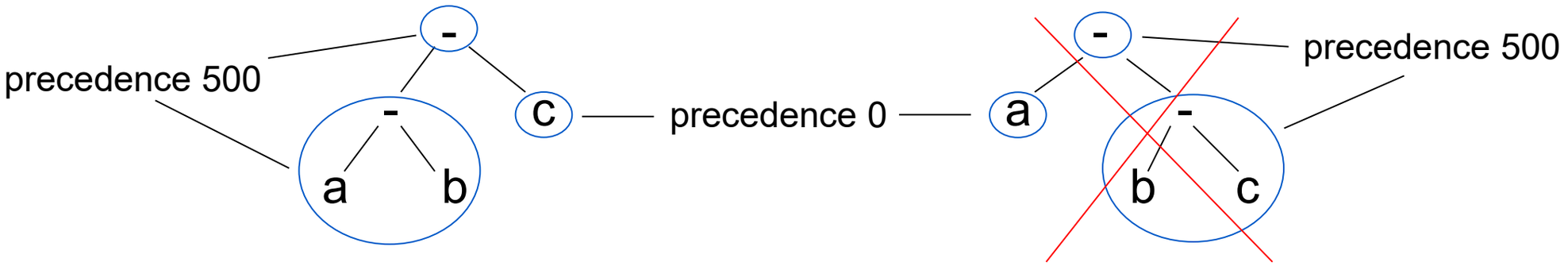
Defining operators in Prolog

For example, + and – , / and * are defined like this:

:-op(500, yfx, [+, -]).

:-op(400, yfx, [*, /]).

Due to the way minus is defined, the expression a-b-c is evaluated (a-b)-c and not a-(b-c).



The precedence is a number between 1 and 1200.

If we define the following operators:

:-op(300,xfx,plays).

:-op(200,xfy,and).

Then **gabi plays football and tennis** = plays(gabi, and(football, tennis)).

Write in the functional form **gabi plays football and tennis and volleyball**.