L6: Reasoning logically - rule-based systems

(Applications of Forward- and Backward-chaining algorithms)

- Review of inference mechanisms
- Rule-based system and its implementation in Java

Seven inference rules for propositional Logic

- R(1) Modus Ponens $$\frac{\alpha \Rightarrow \beta,\ \alpha}{\beta}$$
- R(2) And-Elimination $$\frac{\alpha_1 \wedge \alpha_2 \wedge \ldots \wedge \alpha_n}{\alpha_i}$$
- R(3) And-Introduction $$\frac{\alpha_1,\ \alpha_2,\ \ldots,\ \alpha_n}{\alpha_1 \wedge \alpha_2 \wedge \ldots \wedge \alpha_n}$$
- R(4) Or-Introduction $$\frac{\alpha_i}{\alpha_1 \vee \alpha_2 \vee \ldots \vee \alpha_n}$$
- R(5) Double-Negation Elimination $$\frac{\neg \neg \alpha}{\beta}$$
- R(6) Unit Resolution $$\frac{\alpha \vee \beta,\ \neg \beta}{\alpha}$$
- R(7) Logic connectives: $$\frac{\alpha \vee \beta,\ \neg \beta \vee \gamma}{\alpha \vee \gamma}$$

The three new inference rules

- R (8) Universal Elimination: For any sentence α, variable *v*, and ground term g:

$$\frac{\forall v \ \ \alpha}{\text{SUBST}(\{v/g\}, \alpha)}$$

Ground term is a term that contains no variables.

key	value
x	rose
y	Murderer
…	…
x	…

e. g., *∀ x Likes(x, IceCream)*, we can use the substitute *{x/Rose}* and infer *Like(Rose, IceCream)*.

- R (9) Existential Elimination: For any sentence α, variable *v*, and constant *k* that does not appear elsewhere in the knowledge base:

$$\frac{\exists v \ \ \alpha}{\text{SUBST}(\{v/k\}, \alpha)}$$

e. g., *∃ x Kill(y, Victim)*, we can infer *Kill(Murderer, Victim)*, as long as *Murderer* does not appear elsewhere in the knowledge base.

- R (10) Existential Introduction: For any sentence α, variable *v* that does not occur in α, and ground term g that does occur in α:

e. g., from *Likes(Rose, IceCream)* we can infer *∃ x Likes(x, IceCream)*.

$$\frac{\alpha}{\exists v \ \text{SUBST}(\{g/v\}, \alpha)}$$

Example of proof (証明)

Bob is a buffalo	1. *Buffalo(Bob)* --*f1*
Pat is a pig	2. *Pig(Pat)* --*f2*
Buffaloes run faster than pigs	3. $\forall$ *x, y Buffalo(x)* $\wedge$ *Pig(y)* $\Rightarrow$ *Faster(x,y)* --*r1*

To proof:

Bob runs faster than Pat

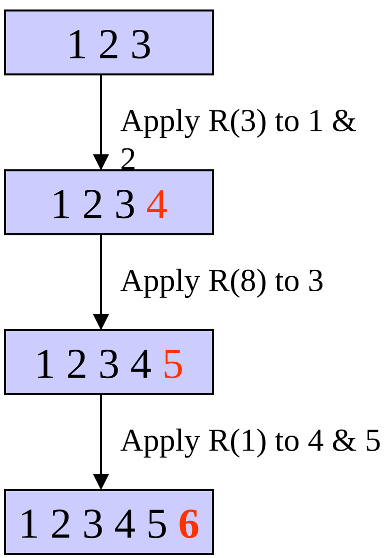
Apply R(3) to *f1* And *f2* (And-Introduction)	4. *Buffalo(Bob)* $\wedge$ *Pig(Pat)* --*f3*
Apply R(8) to *r1* {x/Bob, y/Pat} (Universal-Elimination)	5. *Buffalo(Bob)* $\wedge$ *Pig(Pat)* $\Rightarrow$ *Faster(Bob,Pat)* --*f4*
Apply R(1) to *f3* And *f4* (Inplication-Elimination)	**6**. *Faster(Bob,Pat)* --*f5*

Search with primitive (基本の) inference rules

Operators are inference rules

States are sets of sentences

Goal test checks state to see if it contains query (質問) sentence



R(1) to R(10) are common inference pattern

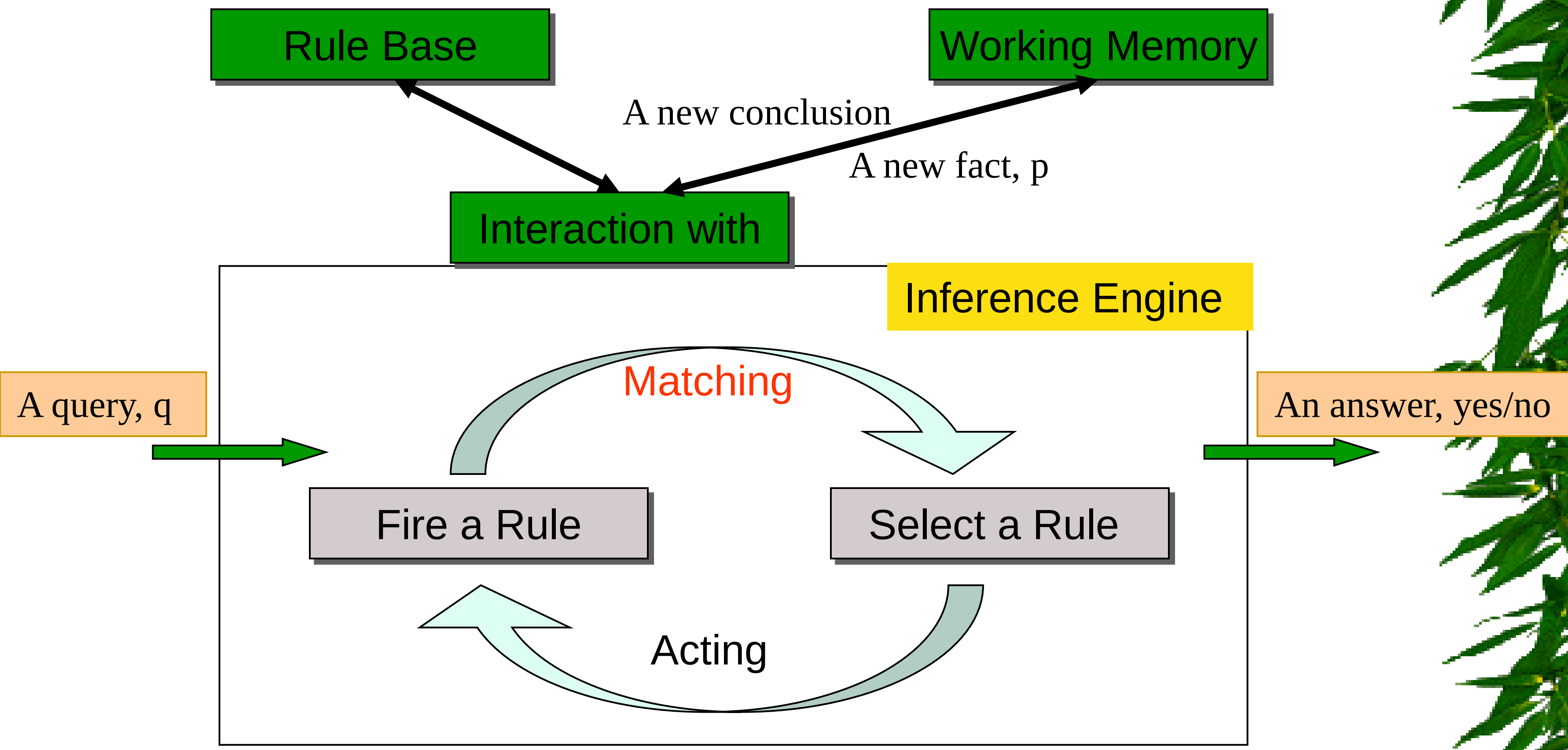
Problem: branching factors are huge, esp. for R(8)

Idea: find a substitution that makes the rule premise match some known facts.

Stored in working memory

Stored in rule base

A Reasoning System



Unify

Unification function, Unify, is to take two atomic sentences *p* and *q* and return a substitution that would make *p* and *q* look the same.

A substitute λ unifies atomic sentences *p* and *q* if $p\lambda = q\lambda$

For example,

p	*q*	λ
Knows(John, x)	*Knows(John, Jane)*	*{x/Jane}*
Knows(John, x)	*Knows(y, OJ)*	*{y/John, x/OJ}*
Knows(John, x)	*Knows(y, Mother(y))*	*{y/John, x/Mother(John)}*

Premise 前提

String matching: string1 = string2

e.g. "rose" = "rose"

"I am Rose" = "I am Rose"

→ if string1.equals(string2)

"I am ?x" = "I am Rose"

"I am ?x" = "?y am Rose"

?

↓

I = ?y

am = am

?x = Rose

e.g. ?x is ?y and ?x
Rose is rose and ?y

↓

?x = Rose

?y = rose

?x = ?y

e.g. husband(father(?x), Mike)
husband(father(John), Mike)

↓

?x = John

```java
public boolean matching(String string1, String string2) {
        // System.out.println(string1);
        // System.out.println(string2);

        // 同じなら成功
        if (string1.equals(string2))
                return true;

        // 各々トークンに分ける
        st1 = new StringTokenizer(string1);
        st2 = new StringTokenizer(string2);

        // 数が異なったら失敗
        if (st1.countTokens() != st2.countTokens())
                return false;

        // 定数同士
        for (int i = 0; i < st1.countTokens();) {
                if (!tokenMatching(st1.nextToken(), st2.nextToken())) {
                        // トークンが一つでもマッチングに失敗したら失敗
                        return false;
                }
        }

        // 最後まで O.K. なら成功
        return true;
}


 boolean tokenMatching(String token1, String token2) {
         // System.out.println(token1+"<->"+token2);
         if (token1.equals(token2))
                 return true;
         if (var(token1) && !var(token2))
                 return varMatching(token1, token2);
         if (!var(token1) && var(token2))
                 return varMatching(token2, token1);
         return false;
 }
```

Forward chaining

If we start with the sentences in the knowledge base and generate new conclusions that in turn can allow more inferences to be made. This is called forward chaining.

TELL

when a new fact *p* is added (told) to the KB

  for each rule such that *p* unifies with a premise

    if the other premises are known

    then add the conclusion to the KB and continue chaining.

- 新しい事実が観測されたときに、事実に最も合う推論を求める
- 事実からスタートして、ルールによって推論結果を得る
- 新たに得られた推論結果は、事実と同じように次の推論に利用できる
- 「AはBである」という事実と、「BならばC」という規則から、「AはCである」という結論を導く推論方式

- Forward chaining is usually used when a new fact is added to the database and we want to generate its consequences.
- It is data driven.

Forward chaining example

Let us add facts r1, r2, r3, f1, f2, f3 in turn into KB.

$\forall$ x, y, z:

r1. Buffalo(x) $\wedge$ *Pig(y)* $\Rightarrow$ *Faster(x,y)*

r2. Pig(y) $\wedge$ *Slug(z)* $\Rightarrow$ *Faster(y,z)*

r3. Faster(x,y) $\wedge$ *Faster(y,z)* $\Rightarrow$ *Faster(x,z)*

f1. Buffalo(Bob) [r1-c1, Bob/x, yes]

f2. Pig(Pat) [r1-c2, Pat/y, yes] → *f4. Faster(Bob, Pat)*

f3. Slug(Steve) [r2-c2, Steve/z, yes]

[r2, f2, f3, Pat/y, Steve/z, yes] → *f5. Faster(Pat, Steve)*

[r3, f4, f5, Bob/x, Pat/y, Steve/z, yes] → *f6. Faster(Bob, Steve)*

Rules defined in the rule base file: CarShop

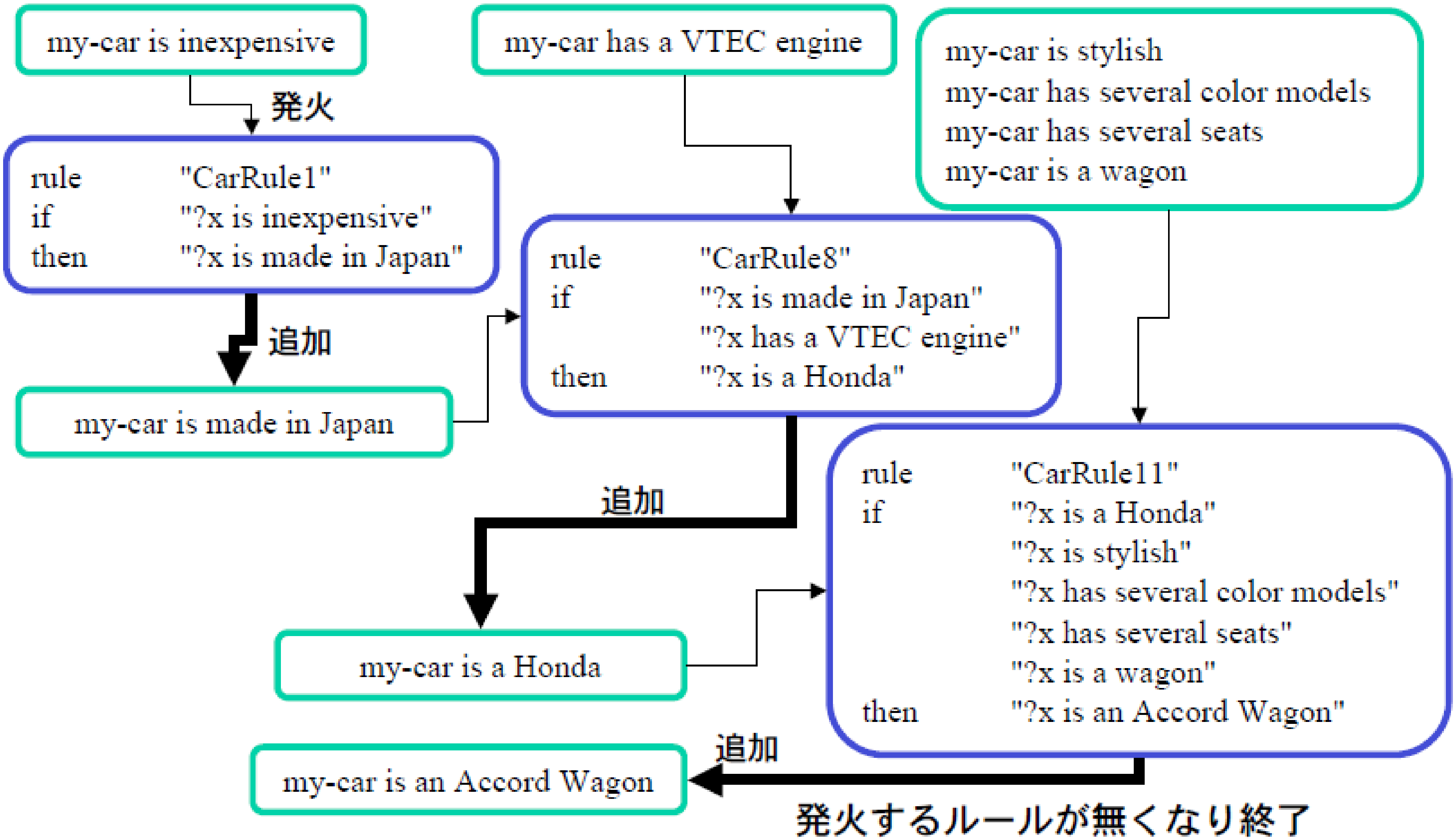
```
rule  "CarRule1"
if    "?x is inexpensive"
then  "?x is made in Japan"
rule  "CarRule2"
if    "?x is small"
then  "?x is made in Japan"
rule  "CarRule3"
if    "?x is expensive"
then  "?x is a foreign car"
rule  "CarRule4"
if    "?x is big"
      "?x needs a lot of gas"
then  "?x is a foreign car"
rule  "CarRule5"
if    "?x is made in Japan"

      "?x has Toyota's logo"
then  "?x is a Toyota"
rule  "CarRule6"
if    "?x is made in Japan"
      "?x is a popular car"
then  "?x is a Toyota"
```

```
rule  "CarRule7"
if    "?x is made in Japan"
      "?x has Honda's logo"
then  "?x is a Honda"
rule  "CarRule8"
if    "?x is made in Japan"
      "?x has a VTEC engine"
then  "?x is a Honda"
rule  "CarRule9“
if    "?x is a Toyota"
      "?x has several seats"
      "?x is a wagon"
then  "?x is a Carolla Wagon"
rule  "CarRule10"
if    "?x is a Toyota"
      "?x has several seats"
      "?x is a hybrid car"
then  "?x is a Prius"
rule  "CarRule11"
if    "?x is a Honda"
      "?x is stylish"
      "?x has several color models"
      "?x has several seats"
      "?x is a wagon"
then  "?x is an Accord Wagon"
```

```
rule  "CarRule12"
if    "?x is a Honda"
      "?x has an aluminium body"
      "?x has only 2 seats"
then  "?x is a NSX"
rule  "CarRule13"
if    "?x is a foreign car"
      "?x is a sports car"
      "?x is stylish"
      "?x has several color models"
      "?x has a big engine"
then  "?x is a Lamborghini Countach"
rule  "CarRule14"
if    "?x is a foreign car"
      "?x is a sports car"
      "?x is red"
      "?x has a big engine"
then  "?x is a Ferrari F50"
rule  "CarRule15"
if    "?x is a foreign car"
      "?x is a good face"
then  "?x is a Jaguar XJ8"
```

Forward Chaining-他の例

rules



Backward chaining

It is to start with something we want to prove, find implication sentences that would allow us to conclude it, and them attempt to establish their premises in turn. This is called backward chaining.

ASK

when a query *q* is asked

if a matching fact *q'* is known, return the unifier

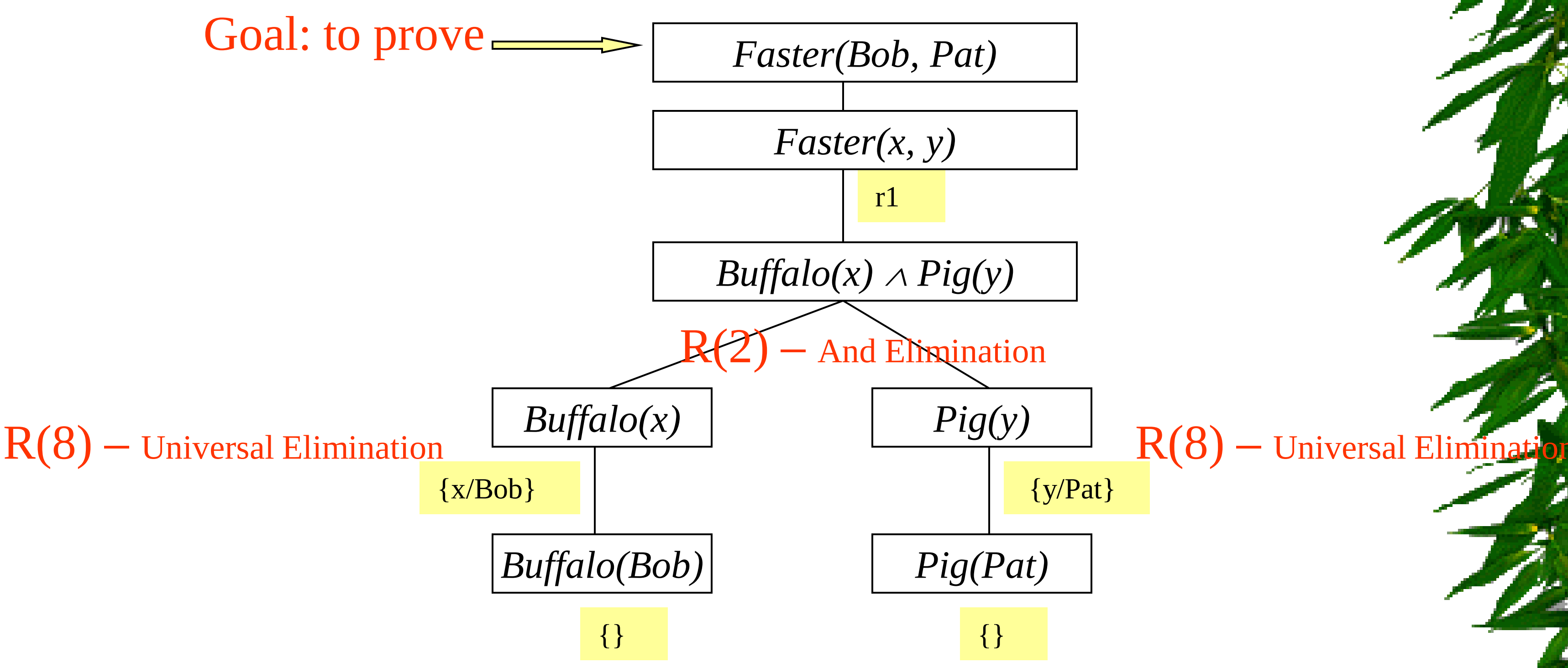
for each rule whose consequent *q'* match *q*

attempt to prove each premise of the rule by backward chaining

・与えられた仮説が、現在のアサーション集合において成り立つかどうかを検証していく推論
・ゴールからスタートする、ゴールが事実の集合にあれば推論成功

Backward chaining example

Bob is a buffalo	1. *Buffalo(Bob)*	--*f1*
Pat is a pig	2. *Pig(Pat)*	--*f2*
Buffaloes run faster than pigs	3. $\forall$ *x, y Buffalo(x)* $\wedge$ *Pig(y)* $\Rightarrow$ *Faster(x,y)*	--*r1*



Rules defined in the rule base file: CarShop

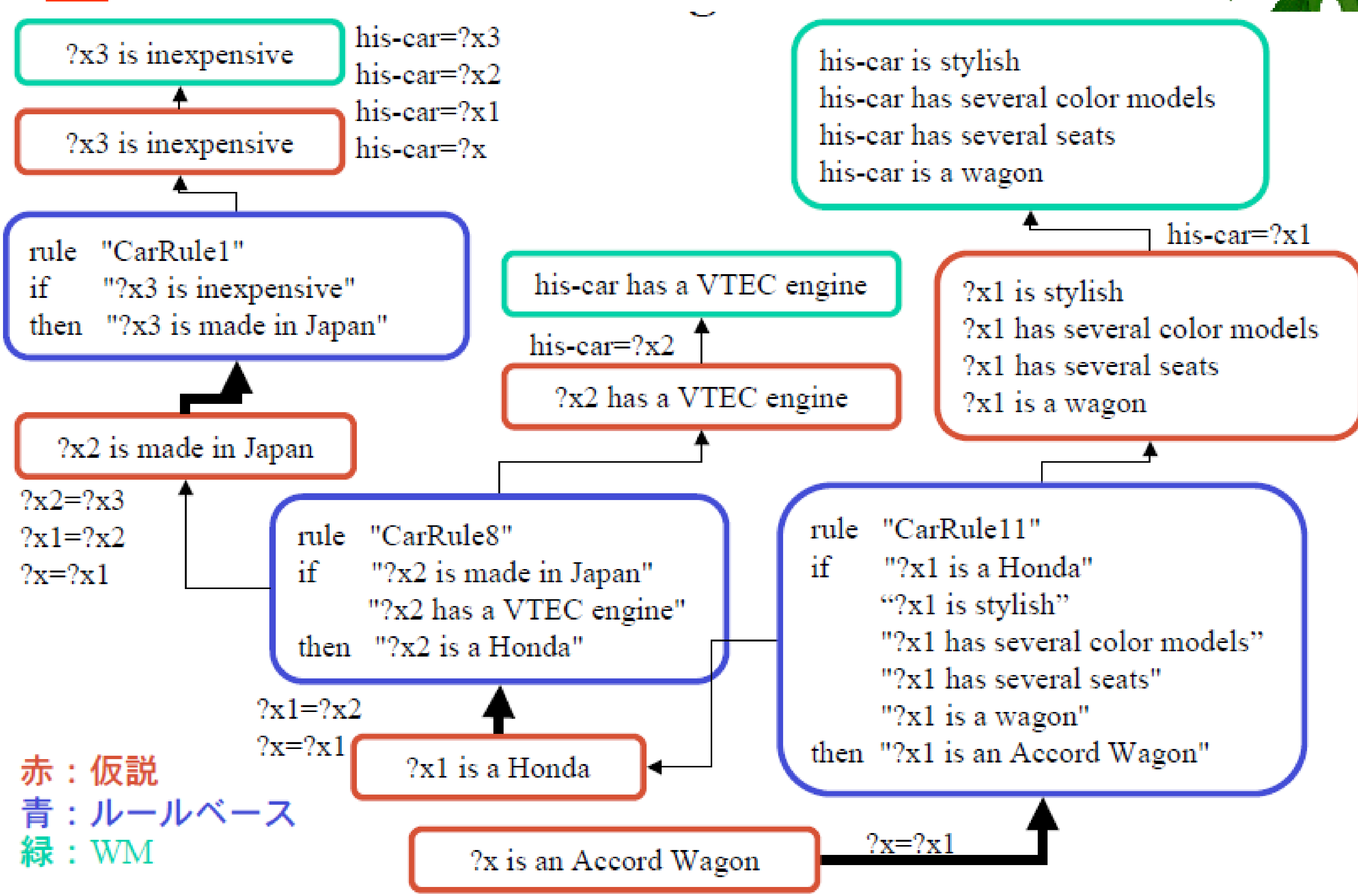
```
rule  "CarRule1"
if    "?x is inexpensive"
then "?x is made in Japan"
rule  "CarRule2"
if    "?x is small"
then "?x is made in Japan"
rule  "CarRule3"
if    "?x is expensive"
then "?x is a foreign car"
rule  "CarRule4"
if     "?x is big"
      "?x needs a lot of gas"
then   "?x is a foreign car"
rule  "CarRule5"
if    "?x is made in Japan"

      "?x has Toyota's logo"
then "?x is a Toyota"
rule  "CarRule6"
if    "?x is made in Japan"
      "?x is a popular car"
then "?x is a Toyota"
```

```
rule  "CarRule7"
if    "?x is made in Japan"
      "?x has Honda's logo"
then "?x is a Honda"
rule  "CarRule8"
if    "?x is made in Japan"
      "?x has a VTEC engine"
then "?x is a Honda"
rule  "CarRule9“
if    "?x is a Toyota"
      "?x has several seats"
      "?x is a wagon"
then "?x is a Carolla Wagon"
rule  "CarRule10"
if    "?x is a Toyota"
      "?x has several seats"
      "?x is a hybrid car"
then "?x is a Prius"
rule  "CarRule11"
if    "?x is a Honda"
      "?x is stylish"
      "?x has several color models"
      "?x has several seats"
      "?x is a wagon"
then "?x is an Accord Wagon"
```

```
rule  "CarRule12"
if    "?x is a Honda"
      "?x has an aluminium body"
      "?x has only 2 seats"
then "?x is a NSX"
rule  "CarRule13"
if    "?x is a foreign car"
      "?x is a sports car"
      "?x is stylish"
      "?x has several color models"
      "?x has a big engine"
then "?x is a Lamborghini Countach"
rule  "CarRule14"
if    "?x is a foreign car"
      "?x is a sports car"
      "?x is red"
      "?x has a big engine"
then "?x is a Ferrari F50"
rule  "CarRule15"
if    "?x is a foreign car"
      "?x is a good face"
then "?x is a Jaguar XJ8"
```

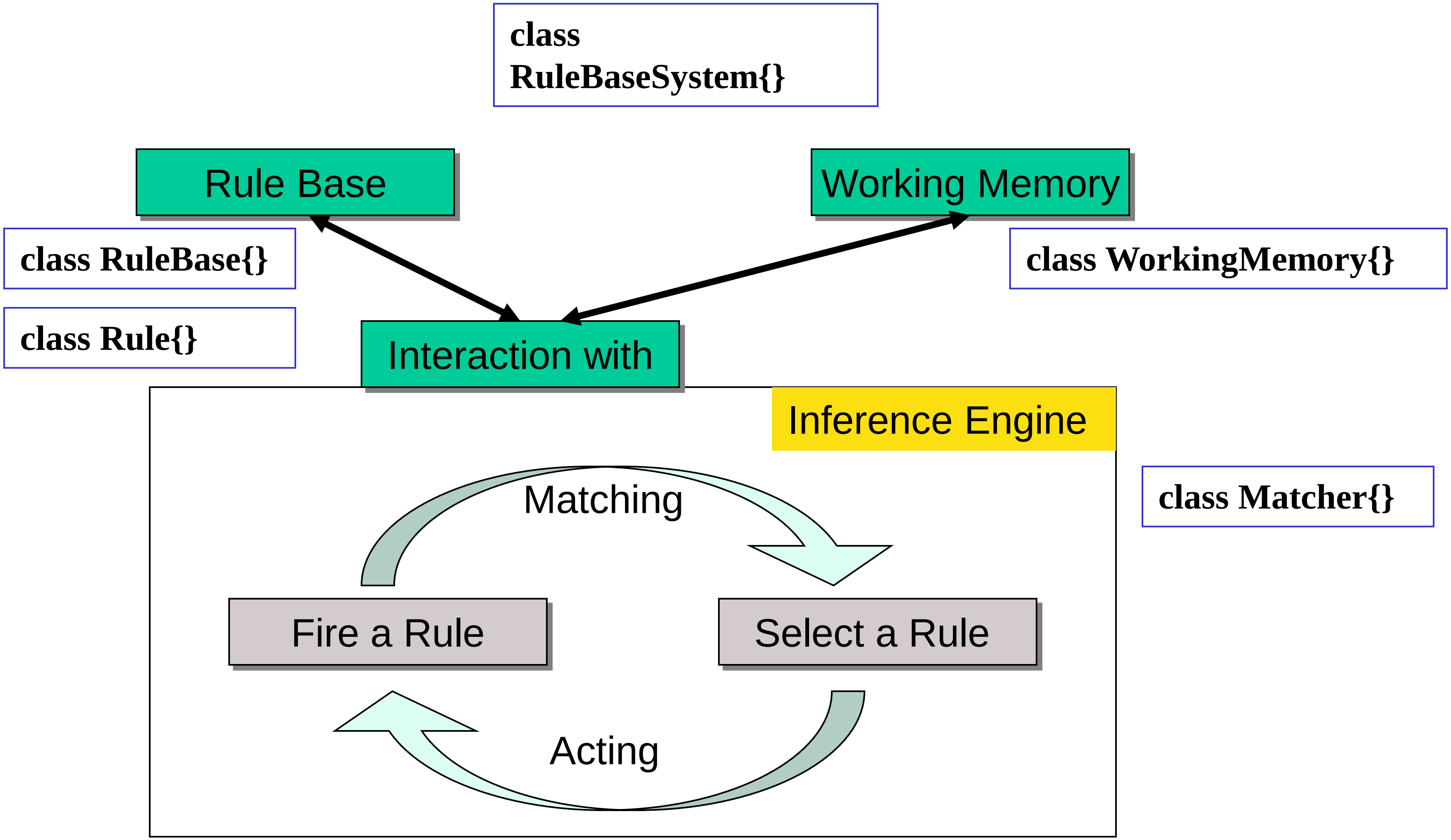
Backward Chaining-他の例

rules



赤：仮説
青：ルールベース
緑：WM

A Rule-base System Architecture



Rule-base System Examples

```
rule      "CarRule1"
if        "?x is inexpensive"
then      "?x is made in Japan"
```

name

antecedents

```
rule      "CarRule4"
if        "?x is big"
          "?x needs a lot of gas"
then      "?x is a foreign car"
```

consequent

```
class WorkingMemory {
  ArrayList<String> assertions;

  WorkingMemory() {
          assertions = new ArrayList<String>();
  }
```

```
RuleBase() {
        fileName = "CarShop.data";
        wm = new WorkingMemory();
        wm.addAssertion("my-car is inexpensive");
        wm.addAssertion("my-car has a VTEC engine");
        wm.addAssertion("my-car is stylish");
        wm.addAssertion("my-car has several color models
        wm.addAssertion("my-car has several seats");
        wm.addAssertion("my-car is a wagon");
        rules = new ArrayList<Rule>();
        loadRules(fileName);
}
```

```
class Rule {
        String name;
        ArrayList<String> antecedents;
        String consequent;

        Rule(String theName, ArrayList<String> theAntecedents,
             String theConsequent) {
                this.name = theName;
                this.antecedents = theAntecedents;
                this.consequent = theConsequent;
        }
```

```
public void addAssertion(String theAssertion){
        System.out.println("ADD:"+theAssertion);
        assertions.addElement(theAssertion);
    }
```

loadRules method

```
private void loadRules(String theFileName) {
        String line;
        try {
                int token;
                f = new FileReader("src/tes/" + theFileName);
                st = new StreamTokenizer(f);
                while ((token = st.nextToken()) != StreamTokenizer.TT_EOF) {
                        switch (token) {
                        case StreamTokenizer.TT_WORD:
                                String name = null;
                                ArrayList<String> antecedents = null;
                                String consequent = null;
                                if ("rule".equals(st.sval)) {
                                        if (st.nextToken() == '"') {
                                                name = st.sval;
                                                st.nextToken();
                                                if ("if".equals(st.sval)) {
                                                        antecedents = new ArrayList<String>();
                                                        st.nextToken();
                                                        while (!"then".equals(st.sval)) {
                                                                antecedents.add(st.sval);
                                                                st.nextToken();
                                                        }
                                                        if ("then".equals(st.sval)) {
                                                                st.nextToken();
                                                                consequent = st.sval;
                                                        }
                                                }
                                        }
                                }
                                rules.add(new Rule(name, antecedents, consequent));
                                break;
                        default:
                                System.out.println(token);
                                break;
                        }
                }
        } catch (Exception e) {
                System.out.println(e);
        }
```

Rule-base System Examples (1)

```
public class RuleBaseSystem {
    static RuleBase rb;
    public static void main(String args[]){
        rb = new RuleBase();
        rb.forwardChain();
    }
}
```

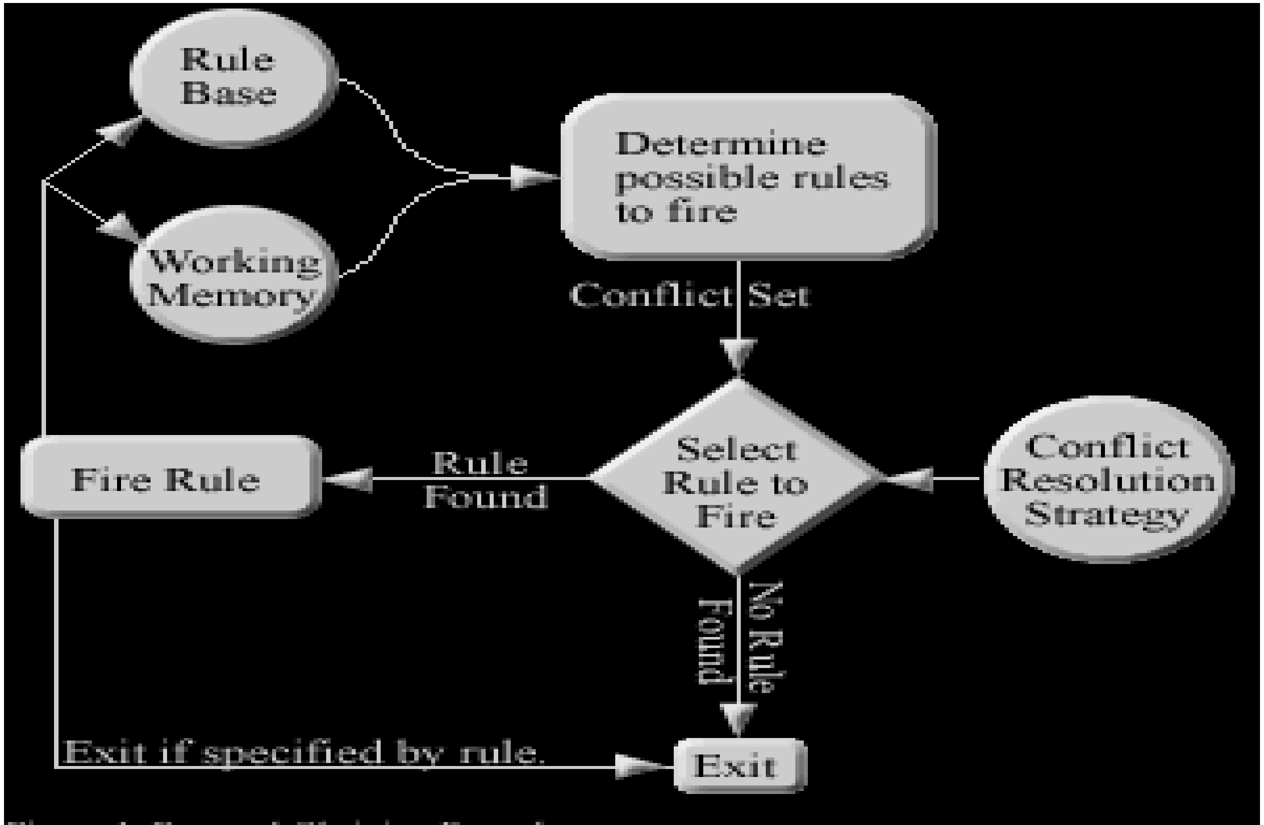


Figure 1: Forward-Chaining Procedure

```java
public void forwardChain() {
	boolean newAssertionCreated;
	// 新しいアサーションが生成されなくなるまで続ける.
	do {
		// 新しいアサーションが生成されたかどうかを保存する変数
		newAssertionCreated = false;
		// ルールの数だけループ
		for (int i = 0; i < rules.size(); i++) {
			// ルールを取り出す
			Rule aRule = rules.get(i);
			// 取り出されたルールの表示
			System.out.println("apply rule:" + aRule.getName());
			// ルールの前件を表示
			ArrayList<String> antecedents = aRule.getAntecedents();
			// ルールの後件を表示
			String consequent = aRule.getConsequent();
			// バインディング情報の取得
			ArrayList<HashMap<String, String>> bindings = wm
					.matchingAssertions(antecedents);

			if (bindings != null) {
				for (int j = 0; j < bindings.size(); j++) {
					// 後件をインスタンシエーション(変数にバインディング情報を当てはめる)
					String newAssertion = instantiate((String) consequent,
							bindings.get(j));
					// ワーキングメモリーになければ成功
					if (!wm.contains(newAssertion)) {
						System.out.println("Success: " + newAssertion);
						// 持っている知識に追加
						wm.addAssertion(newAssertion);
						newAssertionCreated = true;
					}
				}
			}

		}
		System.out.println("Working Memory" + wm);
	} while (newAssertionCreated);
	System.out.println("No rule produces a new assertion");
}
```

matchable() Method

```
public ArrayList<HashMap<String, String>> matchingAssertions(
                ArrayList<String> theAntecedents) {
        ArrayList<HashMap<String, String>> bindings = new ArrayList<HashMap<String, String>>();
        return matchable(theAntecedents, 0, bindings);
}

 private ArrayList<HashMap<String, String>> matchable(
                ArrayList<String> theAntecedents, int n,
                ArrayList<HashMap<String, String>> bindings) {
        // 前件の数だけ繰り返す
        if (n == theAntecedents.size()) {
                return bindings;
        }
        // １つ目
        else if (n == 0) {
                boolean success = false;
                // わかっている知識の数だけループ
                for (int i = 0; i < assertions.size(); i++) {
                        // バインディング情報を保持するハッシュマップ
                        HashMap<String, String> binding = new HashMap<String, String>();
                        // マッチングに成功
                        if ((new Matcher()).matching(theAntecedents.get(n),
                                        assertions.get(i), binding)) {
                                // バインディング情報をハッシュマップに追加
                                bindings.add(binding);
                                success = true;
                        }
                }
                if (success) {
                        return matchable(theAntecedents, n + 1, bindings);
                } else {
                        return null;
                }
        }       . . .
```

```
// ２つ目以降
else {
        boolean success = false;
        // バインディング情報を保持するハッシュマップ
        ArrayList<HashMap<String, String>> newBindings = new ArrayList<HashMap<String, String>>();
        // 得られたバインディング情報の数だけループ
        for (int i = 0; i < bindings.size(); i++) {
                // わかっている知識の数だけループ
                for (int j = 0; j < assertions.size(); j++) {
                        // マッチングに成功
                        if ((new Matcher()).matching(theAntecedents.get(n),
                                        assertions.get(j), bindings.get(i))) {
                                // バインディング情報をハッシュマップに追加
                                newBindings.add(bindings.get(i));
                                success = true;
                        }
                }
        }
        if (success) {
                return matchable(theAntecedents, n + 1, newBindings);
        } else {
                return null;
        }
}
```

前向き推論の解説

はじめに

ソース内に出てくる変数名に使われている単語は以下の意味がある

- Assertion　:わかっている知識(my-car is a wagon等)
- Antecedents:前件（ルールの条件,IFの部分）
- Consequent:後件(ルールから得られる結論, THENの部分)
- Bindings　:接合(知識と前件・後件の変数を結びつける)

前向き推論の途中まで

```
public void forwardChain() {
boolean newAssertionCreated;
// 新しいアサーションが生成されなくなるまで続ける.
do {
        // 新しいアサーションが生成されたかどうかを保存する変数
        newAssertionCreated = false;
        // ルールの数だけループ
        for (int i = 0; i < rules.size(); i++) {
                // ルールを取り出す
                Rule aRule = rules.get(i);
                // 取り出されたルールの表示
                System.out.println("apply rule:" + aRule.getName());
                // ルールの前件を取得
                ArrayList<String> antecedents = aRule.getAntecedents();
                // ルールの後件を取得
                String consequent = aRule.getConsequent();
                // バインディング情報の取得
                ArrayList<HashMap<String, String>> bindings = wm
                                        .matchingAssertions(antecedents);
```

CarRule1

"?x is inexpensive"

"?x is made in Japan"

?x=my-car

前向き推論の続き

my-car is made in Japan

```
if (bindings != null) {
        for (int j = 0; j < bindings.size(); j++) {
                // 後件をインスタンシエーション(変数にバインディング情報を当てはめる)
                String newAssertion = instantiate((String) consequent,bindings.get(j));
                // ワーキングメモリーになければ成功
                if (!wm.contains(newAssertion)) {
                        System.out.println("Success: " + newAssertion);
                        // 持っている知識に追加
                        wm.addAssertion(newAssertion);
                        newAssertionCreated = true;
                }
        }
}
}
System.out.println("Working Memory" + wm);
}
//全てのルールを見て1つでも知識が追加されたら、1から見直す
while (newAssertionCreated);
System.out.println("No rule produces a new assertion");
}
```

ワークングメモリーの中

my-car is inexpensive
my-car has a VTEC engine
:
my-car is made in Japan

matchingAssertionsメソッド

引数は、そのルールの前件で、ルールに含まれる変数を知識と結びつけたものを返す

```
public ArrayList<HashMap<String, String>> matchingAssertions(
ArrayList<String> theAntecedents) {
        //変数と知識を結びつけた　HashMapを保持するリスト
        ArrayList<HashMap<String, String>> bindings = new
        ArrayList<HashMap<String, String>>();
        return matchable(theAntecedents, 0, bindings);
}
```

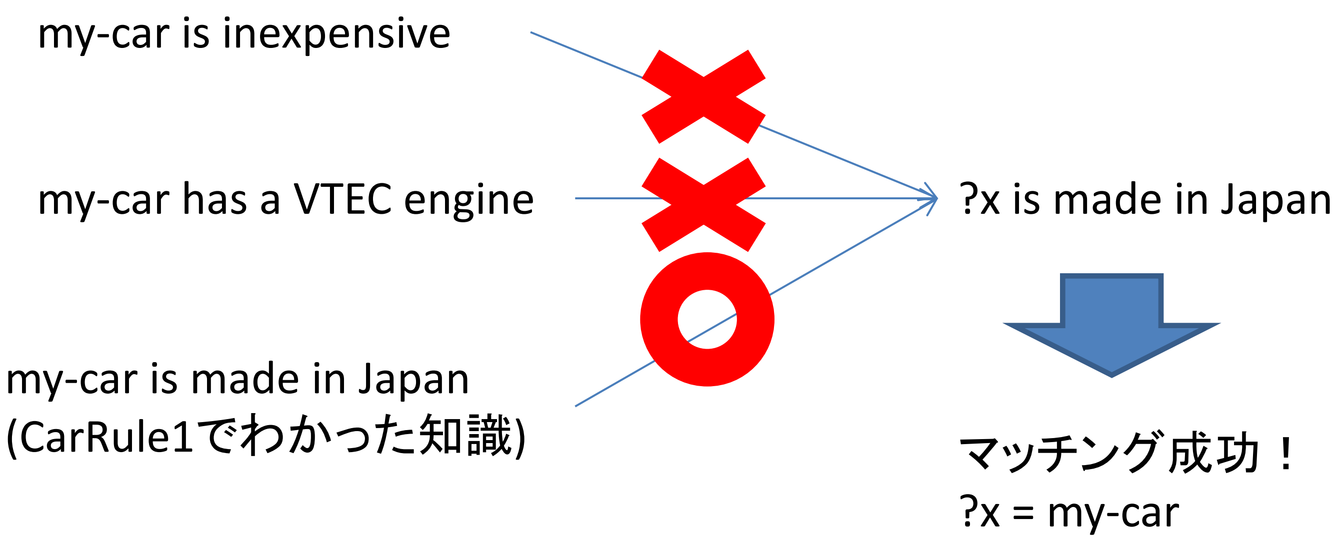
Matchableメソッド

ルールの前件の数だけ持っている知識と繰り返してマッチングを行いバインディング情報を返す

```
// 前件の数だけ繰り返す
if (n == theAntecedents.size()) {
        return bindings;
}
// 1つ目
else if (n == 0) {
    boolean success = false;
    // わかっている知識の数だけループ
    for (int i = 0; i < assertions.size(); i++) {
        // バインディング情報を保持するハッシュマップ
        HashMap<String, String> binding = new HashMap<String, String>();
        // マッチングに成功
        if ((new Matcher()).matching(theAntecedents.get(n),assertions.get(i), binding)) {
            // バインディング情報をハッシュマップに追加
            bindings.add(binding);
            success = true;
        }
    }
    if (success) {
        return matchable(theAntecedents, n + 1, bindings);
    } else {
        return null;
    }
```

１個目のマッチング

- Matcherクラスのmatchingメソッドで知識と前件の照合を行い、バインディング情報を得る
- 例：CarRule8



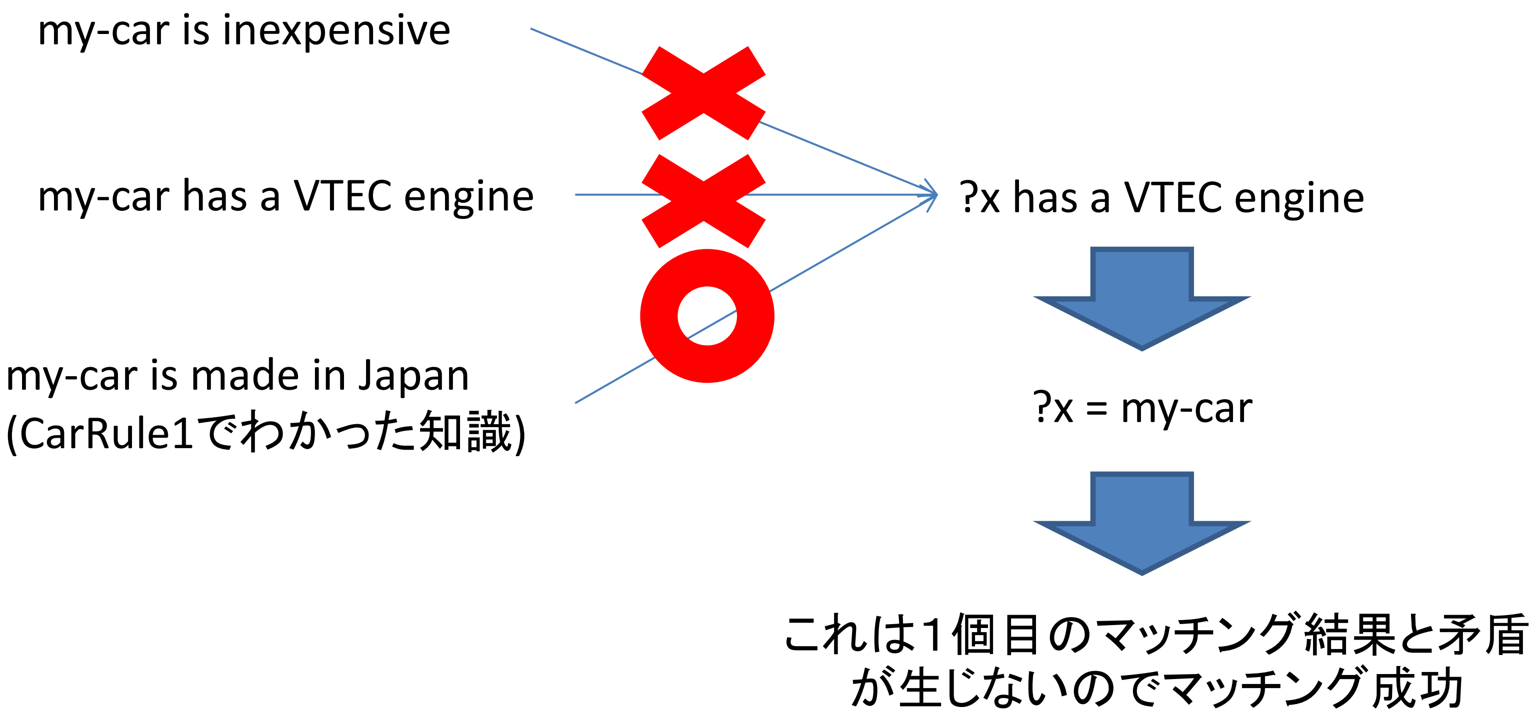
Matchableメソッド

```
else {
    boolean success = false;
    // バインディング情報を保持するハッシュマップ
    ArrayList<HashMap<String, String>> newBindings = new ArrayList<HashMap<String, String>>();
    // 得られたバインディング情報の数だけループ
    for (int i = 0; i < bindings.size(); i++) {
            // わかっている知識の数だけループ
            for (int j = 0; j < assertions.size(); j++) {
                    // マッチングに成功
                    if ((new Matcher()).matching(theAntecedents.get(n),assertions.get(j), bindings.get(i))) {
                            // バインディング情報をハッシュマップに追加
                            newBindings.add(bindings.get(i));
                            success = true;
                    }
            }
    }
    if (success) {
            return matchable(theAntecedents, n + 1, newBindings);
    } else {
            return null;
    }
}
```

前件が複数ある場合は、マッチングに既知のバインディング情報に追加する

２個目以降のマッチング

- １個目の前件と知識の照合で得た?x=my-carというバインディング情報も利用してマッチング



```
rule      "CarRule1"
if        "?x is inexpensive"
then      "?x is made in Japan"

Rule      "CarRule2"
if        "?x is small"
then      "?x is made in Japan"

rule      "CarRule3"
If        "?x is expensive"
then      "?x is a foreign car"

rule      "CarRule4"
if        "?x is big"
          "?x needs a lot of gas"
then      "?x is a foreign car"

Rule      "CarRule5"
If        "?x is made in Japan"
"?x has Toyota's logo"
then      "?x is a Toyota"

rule      "CarRule6"
if        "?x is made in Japan"
          "?x is a popular car"
Then      "?x is a Toyota"
```

```
rule      "CarRule7"
if        "?x is made in Japan"
          "?x has Honda's logo"
then      "?x is a Honda"

rule      "CarRule8"
if        "?x is made in Japan"
          "?x has a VTEC engine"
then      "?x is a Honda"

rule      "CarRule9"
if        "?x is a Toyota"
          "?x has several seats"
          "?x is a wagon"
then      "?x is a Carolla Wagon"

rule      "CarRule10"
if        "?x is a Toyota"
          "?x has several seats"
          "?x is a hybrid car"
then      "?x is a Prius"

rule      "CarRule11"
if        "?x is a Honda"
          "?x is stylish"
          "?x has several color models"
          "?x has several seats"
          "?x is a wagon"
then      "?x is an Accord Wagon"
```

```
rule      "CarRule12"
if        "?x is a Honda"
          "?x has an aluminium body"
          "?x has only 2 seats"
then      "?x is a NSX"

rule      "CarRule13"
if        "?x is a foreign car"
          "?x is a sports car"
          "?x is stylish"
          "?x has several color models"
          "?x has a big engine"
then      "?x is a Lamborghini Countach"

rule      "CarRule14"
if        "?x is a foreign car"
          "?x is a sports car"
          "?x is red"
          "?x has a big engine"
then      "?x is a Ferrari F50"

rule      "CarRule15"
if        "?x is a foreign car"
          "?x is a good face"
then      "?x is a Jaguar XJ8"
```

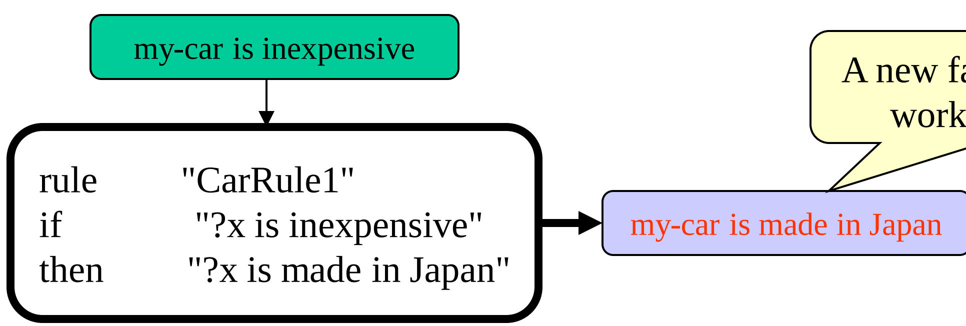
Facts in Working Memory (WM):

his-car is inexpensive
his-car has a VTEC engine
his-car is stylish
his-car has several color models
his-car has several seats
his-car is a wagon

Output:

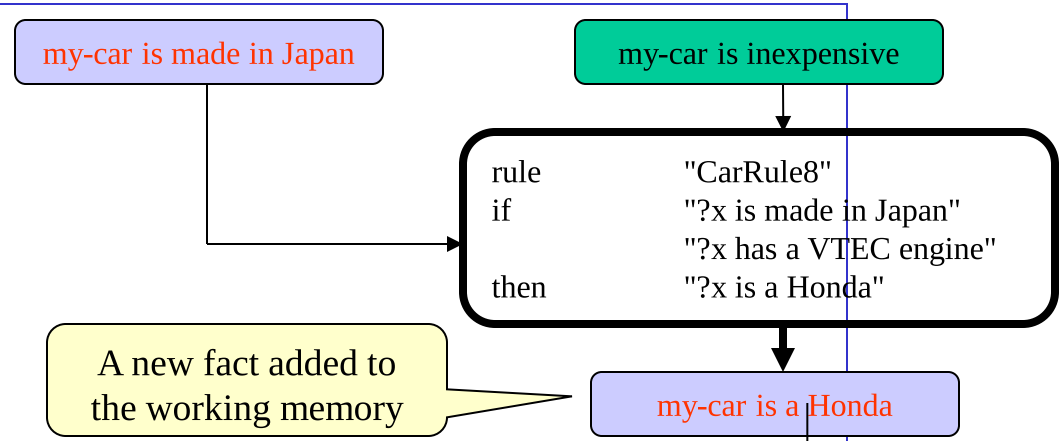
```
% java RuleBaseSystem
% java RuleBaseSystem
ADD:my-car is inexpensive
ADD:my-car has a VTEC engine
ADD:my-car is stylish
ADD:my-car has several color models
ADD:my-car has several seats
ADD:my-car is a wagon
CarRule1 [?x is inexpensive]->?x is made in Japan
CarRule2 [?x is small]->?x is made in Japan
CarRule3 [?x is expensive]->?x is a foreign car
CarRule4 [?x is big, ?x needs a lot of gas]->?x is a foreign car
CarRule5 [?x is made in Japan, ?x has Toyota's logo]->?x is a Toyota
CarRule6 [?x is made in Japan, ?x is a popular car]->?x is a Toyota
CarRule7 [?x is made in Japan, ?x has Honda's logo]->?x is a Honda
CarRule8 [?x is made in Japan, ?x has a VTEC engine]->?x is a Honda
CarRule9 [?x is a Toyota, ?x has several seats, ?x is a wagon]->?x is a Carolla Wagon
CarRule10 [?x is a Toyota, ?x has several seats, ?x is a hybrid car]->?x is a Prius
CarRule11 [?x is a Honda, ?x is stylish, ?x has several color models, ?x has several seats, ?x is a wagon]->?x is an Accord Wagon
CarRule12 [?x is a Honda, ?x has an aluminium body, ?x has only 2 seats]->?x is a NSX
CarRule13 [?x is a foreign car, ?x is a sports car, ?x is stylish, ?x has several color models, ?x has a big engine]->?x is a Lamborghini Countach
CarRule14 [?x is a foreign car, ?x is a sports car, ?x is red, ?x has a big engine]->?x is a Ferrari F50
CarRule15 [?x is a foreign car, ?x is a good face]->?x is a Jaguar XJ8
apply rule:CarRule1
Success: my-car is made in Japan
ADD:my-car is made in Japan
apply rule:CarRule2
apply rule:CarRule3
apply rule:CarRule4
apply rule:CarRule5
apply rule:CarRule6
apply rule:CarRule7
```

Initial facts in the working memory

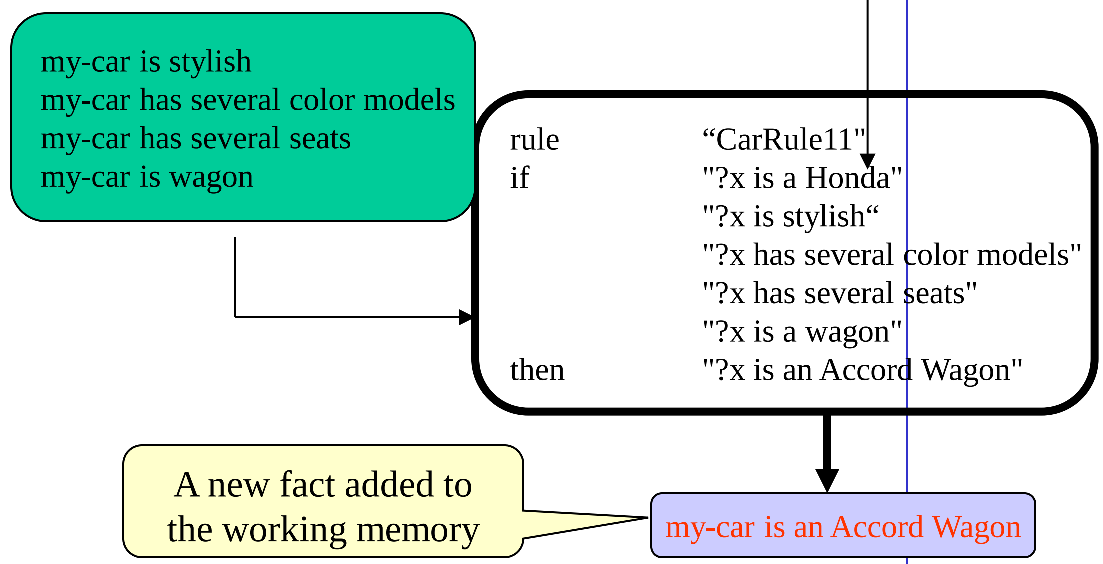


A new fact added to the working memory

apply rule:CarRule8
Success: my-car is a Honda
ADD:my-car is a Honda
apply rule:CarRule9
apply rule:CarRule10
apply rule:CarRule11
Success: my-car is an Accord Wagon
ADD:my-car is an Accord Wagon
apply rule:CarRule12
apply rule:CarRule13
apply rule:CarRule14
apply rule:CarRule15

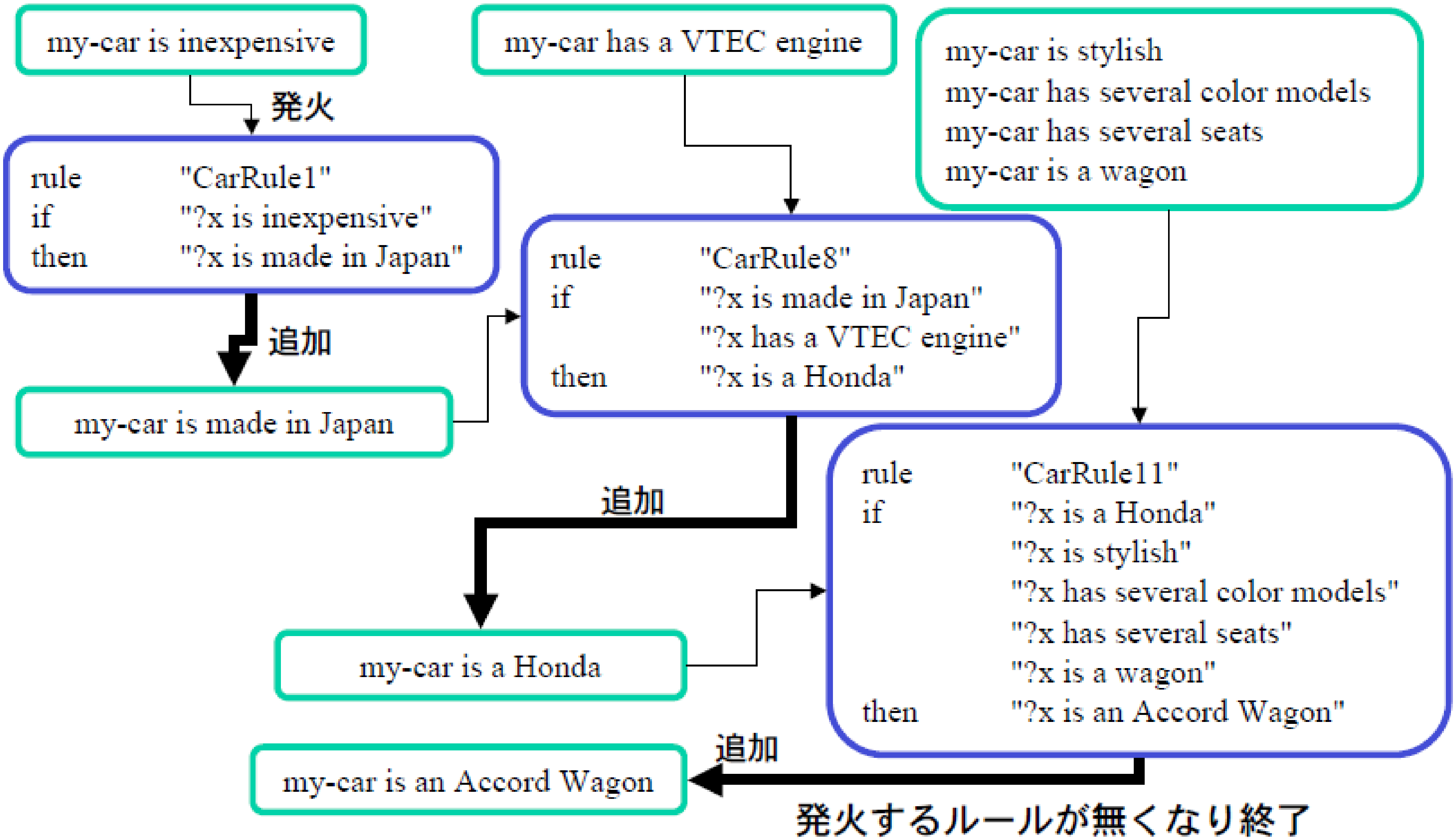


Working Memory[my-car is inexpensive, my-car has a VTEC engine, my-car is stylish, my-car has several color models, my-car has several seats, my-car is a wagon, my-car is made in Japan, my-car is a Honda, my-car is an Accord Wagon]
apply rule:CarRule1
apply rule:CarRule2
apply rule:CarRule3
apply rule:CarRule4
apply rule:CarRule5
apply rule:CarRule6
apply rule:CarRule7
apply rule:CarRule8
apply rule:CarRule9
apply rule:CarRule10
apply rule:CarRule11
apply rule:CarRule12
apply rule:CarRule13
apply rule:CarRule14
apply rule:CarRule15



Working Memory[my-car is inexpensive, my-car has a VTEC engine, my-car is stylish, my-car has several color models, my-car has several seats, my-car is a wagon, my-car is made in Japan, my-car is a Honda, my-car is an Accord Wagon]
No rule produces a new assertion

Forward Chaining



Exercise – AI programming

1. Input program RuleBaseSystem.java, and a rule-base file CarShop.txt understand them, run the program, and check the output.
2. Make some modifications to RuleBaseSystem.java (data file name: Outruns.data, rules in the data file, and writing the initial content (facts) in the working memory in a file, wm.txt) so that the program can run for the following case.

Bob is a buffalo	1. *Buffalo(Bob)*
Pat is a pig	2. *Pig(Pat)*
Buffaloes outrun pigs	3. $\forall$ *x, y Buffalo(x)* $\wedge$ *Pig(y)* $\Rightarrow$ *Faster(x,y)*

Bob outruns Pat

Apply (3) to 1 And 2	4. *Buffalo(Bob)* $\wedge$ *Pig(Pat)*
Apply (8) to 3 {x/Bob, y/Pat}	5. *Buffalo(Bob)* $\wedge$ *Pig(Pat)* $\Rightarrow$ *Faster(Bob,Pat)*
Apply (1) to 4 And 5	6. *Faster(Bob,Pat)*

Output:

```
BootStrap RuleBaseSystem
AppAccelerator(tm) 1.2.010 for Java (JDK 1.2), x86 version.
Copyright (c) 1997-1999 Inprise Corporation. All Rights Reserved.
ADD:Buffalo( Bob )
ADD:Pig( Pat )
r3 [?X, ?Y]->?X & ?Y
f3 [Buffalo( ?X ), Pig( ?Y )]->Faster( ?X , ?Y )
apply rule:r3
apply rule:f3
Success: Faster( Bob , Pat )
ADD:Faster( Bob , Pat )
Working Memory[Buffalo( Bob ), Pig( Pat ), Faster( Bob , Pat )]
apply rule:r3
apply rule:f3
Working Memory[Buffalo( Bob ), Pig( Pat ), Faster( Bob , Pat )]
No rule produces a new assertion

アプリケーションを停止するには Ctrl+C を押してください
```

Home work

Write a report includes
- Front page (name, id)
- About the report
  - exercise problem statement
  - Where you make effort
- Source program
- Execution screen shot

The submission deadline: 2015/11/05