FAF.FIA16.1 -- Artificial Intelligence Fundamentals

> **Lab 1:** Expert Systems \ **Performed by:** Gherman Artiom, group FAF-193 \
> **Verified by:** Mihail Gavrilita, asist. univ.

Imports and Utils

In [18]:
```
# !python -m venv venv
%pip install regex
```

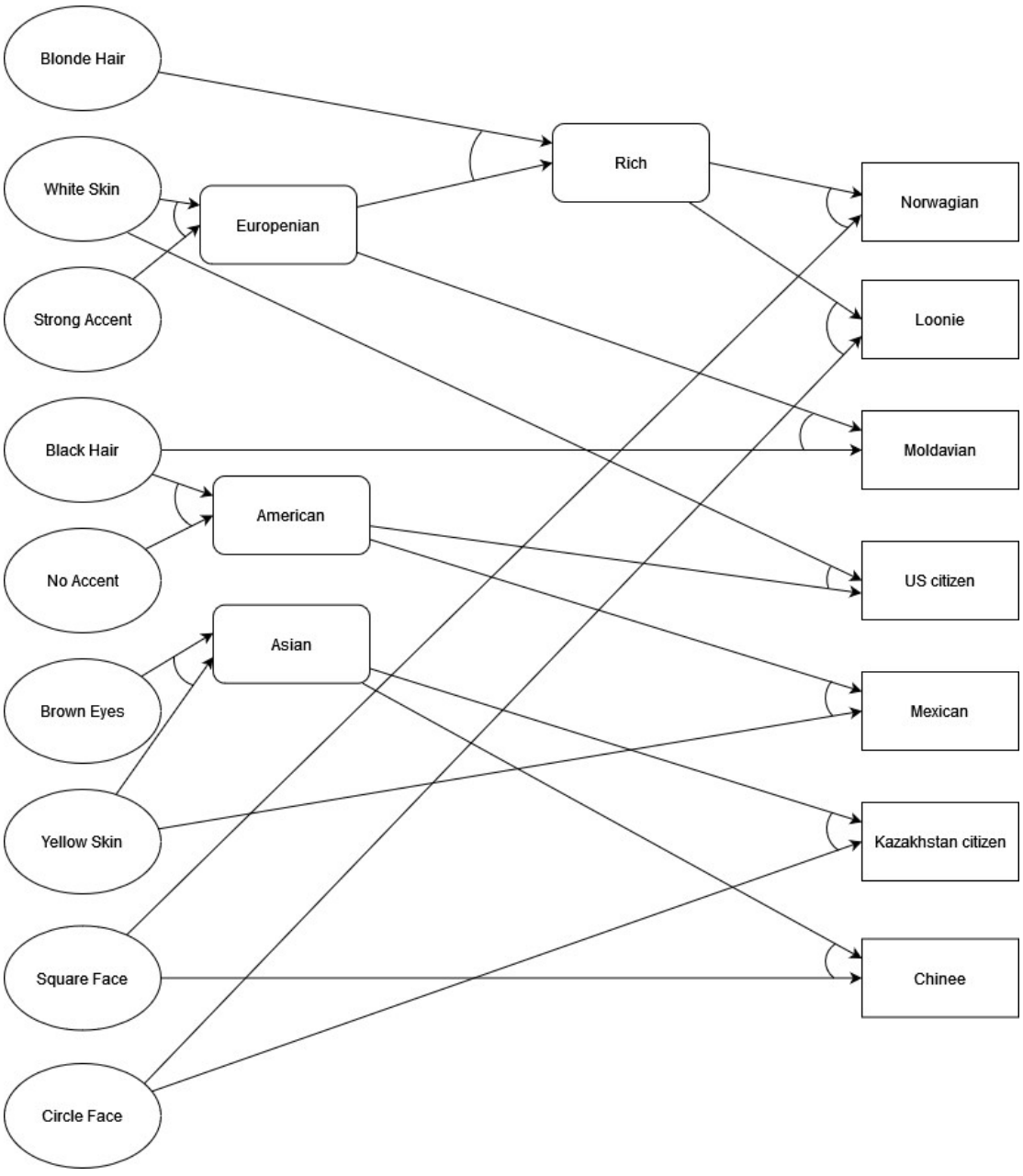
```
Collecting regex
  Using cached regex-2022.10.31-cp39-cp39-win_amd64.whl (267 kB)
Installing collected packages: regex
Successfully installed regex-2022.10.31
Note: you may need to restart the kernel to use updated packages.
WARNING: You are using pip version 21.2.3; however, version 23.0 is available.
You should consider upgrading via the 'C:\Users\Legion\AppData\Local\Programs\Py
thon\Python39\python.exe -m pip install --upgrade pip' command.
```

In [2]:
```
from rules import CITIZEN_RULES, CITIZEN_DATA
from production import run_conditions, backward_chain, setFacts
from production import IF, AND, THEN, OR, DELETE, NOT, FAIL
from production import match, instantiate, FactSet
from collections import defaultdict
from IPython import display
```

Task 1 -- Define 5 types of tourist that visit Luna-City. Draw Goal Tree.

There defined 6 types of toursist, plus Loonie

- Norwagian toursit
- Moldavian tourist
- Loonie
- US citizen
- Mexican
- Kazakhstan citizen
- Chinee



Task 2 -- Implement the rules from the defined tree.

In [3]:
```
CITIZEN_RULES = (
    IF( AND("(?x) has strong accent",
            "(?x) has white skin"),
       THEN("(?x) is from Europe") ),
    IF( AND("(?x) has black hair",
            "(?x) has no accent"),
       THEN("(?x) is from America") ),
    IF( AND("(?x) has brown eyes",
            "(?x) has yellow skin"),
       THEN("(?x) is from Asia") ),
    IF( AND("(?x) has blonde hair",
            "(?x) is from Europe"),
       THEN("(?x) is rich") ),
    IF( AND("(?x) is rich",
            "(?x) has square face"),
       THEN("(?x) is Norwagian") ),
    IF( AND("(?x) is rich",
            "(?x) has circle face"),
       THEN("(?x) is Loonie!") ),
    IF( AND("(?x) is from Europe",
            "(?x) has black hair"),
       THEN("(?x) is Moldavian") ),
    IF( AND("(?x) is from America",
            "(?x) has white skin"),
       THEN("(?x) is US citizen") ),
    IF( AND("(?x) is from America",
            "(?x) has yellow skin"),
       THEN("(?x) is Mexican") ),
    IF( AND("(?x) is from Asia",
            "(?x) has circle face"),
       THEN("(?x) is Kazakhstan citizen") ),
    IF( AND("(?x) is from Asia",
            "(?x) has square face"),
       THEN("(?x) is Chinee") ),
)

CITIZEN_DATA = (
    "mark has white skin",
    "mark has strong accent",
    "mark has black hair",
    "mark has square face",
    "amma has no accent",
    "amma has black hair",
    "amma has white skin",
    "jimmy has white skin",
    "jimmy has strong accent",
    "jimmy has blonde hair",
    "jimmy has circle face",
    "claus has black hair",
    "claus has no accent",
    "claus has yellow skin",
    "nurlan has brown eyes",
    "nurlan has yellow skin",
    "nurlan has circle face",
    "jackie has brown eyes",
    "jackie has yellow skin",
    "jackie has square face",
)
```

Task 3 -- Check if Forward Chaining Algorithm for provided Goal Tree

In [4]:
```
print(run_conditions(CITIZEN_RULES, CITIZEN_DATA))
```

```
('amma has black hair', 'amma has no accent', 'amma has white skin', 'amma is US
citizen', 'amma is from America', 'claus has black hair', 'claus has no accent',
'claus has yellow skin', 'claus is Mexican', 'claus is from America', 'jackie ha
s brown eyes', 'jackie has square face', 'jackie has yellow skin', 'jackie is Ch
inee', 'jackie is from Asia', 'jimmy has blonde hair', 'jimmy has circle face',
'jimmy has strong accent', 'jimmy has white skin', 'jimmy is Loonie!', 'jimmy is
from Europe', 'jimmy is rich', 'mark has black hair', 'mark has square face', 'm
ark has strong accent', 'mark has white skin', 'mark is Moldavian', 'mark is fro
m Europe', 'nurlan has brown eyes', 'nurlan has circle face', 'nurlan has yellow
skin', 'nurlan is Kazakhstan citizen', 'nurlan is from Asia')
```

Task 4 -- Implement Backward Chaining Algorithm for the Goal Tree

Backward Chaining just goes through each consequent and verify if it doesn't has its own consequents. If it is not - the point we wanted to achieve is saved in list of facts as other intermediat groups and characteristics that are just antecedents.

In [5]:
```
def setFacts(rules, hypothesis):
    facts = FactSet()
    backward_chain(rules, hypothesis, facts)
    results = set(facts.set_of_facts)
    cons = set(facts.set_of_consequents)
    facts.clearAll()

    formattedFacts = set()

    formattedFacts = set(instantiate(result, values_dict)
                         for con in cons if (values_dict := match(con, hypothesi
                         for result in results)

    return formattedFacts

def backward_chain(rules, hypothesis, fact_set, verbose=False):

    if not isinstance(hypothesis, list):
        hypothesis = [hypothesis]

    for hypo in hypothesis:
        l = [match(list(condition.consequent())[0], hypo) != None for condition
        try:
            condition_index = l.index(True)
            fact_set.addElement(hypo)
            fact_set.addConsequent(list(rules[condition_index].consequent())[0])
        except ValueError:
            fact_set.addElement(hypo)
            return None
        for ant in list(rules[condition_index].antecedent()):
            backward_chain(rules, ant, fact_set)

    return None

enciclopedia = setFacts(CITIZEN_RULES, "nurlan is Kazakhstan citizen") # Put her
print(*(element for element in enciclopedia), sep="\n")
```

```
nurlan has brown eyes
nurlan is Kazakhstan citizen
nurlan has yellow skin
nurlan is from Asia
nurlan has circle face
```

Task 5 and 7 -- Implement a system for generating questions from the Goal Tree

Occurance class

This class was create to store state of each rule. There two main dictionaries: `occurances` and `probability`. The `occurances` dictionary stores info about each rule. It can have three states: `None` - if we haven't met this rule yet, `True` - if the value was marked as the fact and `False` - if the rule was declined. Dictionary called `probability` stores the info how valuable the rule is. Probability depends on how many times rule appears in other `IF`s and depends how big the group of the same type is.

Questions

Questions to ask appear depend on the biggest value in `probability` dictionary. The order of asking questions may vary with growing number of rules.

In [9]:
```
class Occurance:
    def __init__(self, citizen_rules):
        self.citizen_rules = citizen_rules
        self.occurances = dict()
        self.all_rules = set()
        self.probability = dict()
        self.total_length = sum(len(list(rule.antecedent())) for rule in self.ci
        self.consequents = set()
        self.type_counts = defaultdict(int)
        self.questions_to_ask = []
        self.antecedents = set()

        self.fullFill()
        self.findConsequents()
        self.findTypes()
        self.calculateProbability()
        self.calculateEffectivness()
        self.findBestCharacteristics()


    def findConsequents(self):
        for rule in self.citizen_rules:
            self.consequents.add(list(rule.consequent())[0])
        self.all_rules -= self.consequents


    def findAntecedents(self):
        for rule in self.citizen_rules:
            self.antecedents.add(list(rule.antecedent())[0])



    def clear(self):
        self.occurances = {rule: None for rule, _ in self.occurances.items()}


    def fullFill(self):
        for rule in self.citizen_rules:
            self.all_rules |= set(list(rule.antecedent()))

        self.occurances = {rule: None for rule in self.all_rules}


    def setOccured(self, element):
        self.occurances[element] = True


    def setNeedless(self, element):
        self.occurances[element] = False


    def calculateProbability(self):
        self.probability = dict()
        for rule in self.all_rules:
            if self.occurances[rule] is False or self.occurances[rule] is True:
                self.probability[rule] = 0
                continue
            rule_probability_counter = sum(
                1 for citizen_rule in self.citizen_rules
                if rule in citizen_rule.antecedent())

            self.probability[rule] = self.findPercent(rule_probability_counter)
```

```
def findPercent(self, occurancy_times):
    return round(occurancy_times / self.total_length, 3)


def findTypes(self):
    for rule in self.all_rules:
        if self.occurances[rule] == False:
            continue
        characteristic_type = rule.split()[-1]
        self.type_counts[characteristic_type] += 1

    for key, value in self.type_counts.items():
        self.type_counts[key] = round(value / len(self.all_rules) / 10, 3)


def updateOccurance(self):
    for rule, res in self.occurances.items():
        if res is True:
            for key, value in self.occurances.items():
                if value is None and key.split()[-1] == rule.split()[-1]:
                    self.occurances[key] = False


def calculateEffectivness(self):
    for key, value in self.probability.items():
        last_word = key.split()[-1]
        if self.occurances[key] == False or self.occurances[key] == True:
            continue
        self.probability[key] = round(value + self.type_counts.get(last_word


def findBestCharacteristics(self):
    self.questions_to_ask = []
    max_value = max(self.probability.values())
    max_rules = [k for k, v in self.probability.items() if v == max_value an

    max_key = max((k.split()[-1], k) for k in max_rules)[1].split()[-1]

    self.questions_to_ask = [rule for rule in max_rules if max_key in rule]

# Task 5
def makeQuestion(self):
    # questions = [questions[0]]
    output_strings = {
        "title":"",
        "question": "",
        "variants_of_answers": "",
        "type":"",
    }

    if len(self.questions_to_ask) == 1:
        output_strings["title"] = "Answer yes or no:"
        output_strings["question"] = self.questions_to_ask[0]
        output_strings["type"] = "s"

    else:
        output_strings["title"] = "Choose one: "
        output_strings["variants_of_answers"] = []
        output_strings["type"] = "m"
        for question in self.questions_to_ask:
            output_strings["variants_of_answers"].append(question)
```

```
        return output_strings


    def askQuestion(self, output, name):
        yeses = ["y", "ye", "yeah", "yep", "YES", "Y", "yes"]
        nos = ["n", "no", "NO", "N", "NOPE", "nope"]

        from production import populate

        if output["type"] == "s":
            print("Answer yes/no:\n")
            # Task 7
            question = "Does " + populate(output["question"], {"x":name}) + "?"
            answer = input(f"{question}\nYour answer is: ")

            if answer in yeses:
                self.setOccured(output["question"])
            elif answer in nos:
                self.setNeedless(output["question"])

        else:
            print("Answer with number of variant:\n")
            # Task 7
            question = [f"{i + 1}. Does " + populate(e, {"x":name}) + "?" for i,

            try:
                answer = int(input(("\n".join(question) + "\nYour answer is: ")))

                for element in output["variants_of_answers"]:
                    if element != output["variants_of_answers"][answer - 1]:
                        self.setNeedless(element)
                    else:
                        self.setOccured(element)
                    self.probability.pop(element)
            except ValueError:
                for element in output["variants_of_answers"]:
                    self.setNeedless(element)
                    self.probability.pop(element)


    def start(self, name):
        from production import run_conditions, instantiate, match

        result = ()
        answer_is_found = False
        rules_forward = set()

        self.fullFill()
        self.findConsequents()
        self.findAntecedents()

        while True:
            self.findTypes()
            self.calculateProbability()
            self.calculateEffectivness()
            self.findBestCharacteristics()
            print(30*"-")
            self.askQuestion(self.makeQuestion(), name)
            self.updateOccurance()

            for fact in self.occurances:
                if self.occurances[fact] is True:
                    rules_forward.add(instantiate(fact, {"x":name}))
```

```
            result = run_conditions(CITIZEN_RULES, rules_forward)

            # print(result)

            for element in result:
                if element.replace(name, "(?x)") not in self.antecedents and ele
                    print("Match found: " + instantiate(element, {"x":name}) + "
                    answer_is_found = True
                    break

            if answer_is_found:
                break


if __name__ == "__main__":
    occ = Occurance(CITIZEN_RULES)
    occ.start("mark")
```

```
------------------------------
Answer with number of variant:

1. Does mark has yellow skin?
2. Does mark has white skin?
Your answer is: 2
------------------------------
Answer yes/no:

Does mark has black hair?
Your answer is: yes
------------------------------
Answer with number of variant:

1. Does mark has square face?
2. Does mark has circle face?
Your answer is: i dpn't know
------------------------------
Answer with number of variant:

1. Does mark has no accent?
2. Does mark has strong accent?
Your answer is: 2
Match found: mark is Moldavian!!!!!
```

Task 6 Wrap up everything in an interactive Expert System

To user gives posibility to choose if he wants the system to guess his tourist (akinator) or to find all the characteristics that tourist should have.

In [10]:
```
print("\nWelcome to Detecting Tourist System!")

while True:
    print(60*"*")
    print("\nWhat I can help you with? Choose by variant of answer:")
    print("1. Match tourist by description\n2. Give info about some type of tour
    answer = int(input())
    print(60*"-")


    if answer == 1:
        akinator = Occurance(CITIZEN_RULES)
        tourist_name = input("Enter the name of the tourist: ")
        akinator.start(tourist_name)

    elif answer == 2:
        tourist_type = input("Enter what type of tourist you want to achieve (ex
        print("\nFacts what should toursist has: \n")
        enciclopedia = setFacts(CITIZEN_RULES, tourist_type)
        print(*(element for element in enciclopedia), sep="\n")

    else:
        break
```

```
Welcome to Detecting Tourist System!
************************************************************

What I can help you with? Choose by variant of answer:
1. Match tourist by description
2. Give info about some type of tourist

1
------------------------------------------------------------
Enter the name of the tourist: lena
------------------------------
Answer with number of variant:

1. Does lena has yellow skin?
2. Does lena has white skin?
Your answer is: 1
------------------------------
Answer yes/no:

Does lena has black hair?
Your answer is: yes
------------------------------
Answer with number of variant:

1. Does lena has square face?
2. Does lena has circle face?
Your answer is: i don knwo
------------------------------
Answer with number of variant:

1. Does lena has no accent?
2. Does lena has strong accent?
Your answer is: 1
Match found: lena is Mexican!!!!!
************************************************************

What I can help you with? Choose by variant of answer:
1. Match tourist by description
2. Give info about some type of tourist

2
------------------------------------------------------------
Enter what type of tourist you want to achieve (ex: mark is Moldavian): ion is L
oonie!

Facts what should toursist has:

ion has strong accent
ion has blonde hair
ion is Loonie!
ion has white skin
ion is from Europe
ion is rich
ion has circle face
************************************************************

What I can help you with? Choose by variant of answer:
1. Match tourist by description
2. Give info about some type of tourist

3
------------------------------------------------------------
```

Conclusions:

There was done laboratory work where we had to create Expert Systems. There is posibility to choose akinator or to open encyclopedia. The main problem was to find the best question that can be asked depending on the answers of the user. It can be found more effective way to calculate the best answers.

Bibliography:

1. What is goal tree? - hohmannchris
2. Forward Chaining Algorithm. - Engati
3. Backward Chaining Algorithm. - Section IO
4. Backward Chaining Example. - Education 4u