# CS8803
# Knowledge Based AI
# Project3

**Puyan Lotfi**

## Implementation:

I used the same code from my project 2. In project 2 I had a grammar that I constructed with yacc, I also had some global maps and lists that I used to store scripts and facts and events. The grammar was used to parse events and load them into memory as Frame objects, and it was also used to parse entries of scripts and load them into memory as Frame objects. I isolated Frames dealing with scripts from frames dealing with events in different collections.

The way my project 2 worked was to do some pattern matching on the scripts and find a collection of events that matched, finally it would do some statistical guessing of the actors involved in the chain of events so that it can take a pattern like “Intension of <action>: go-to-store” and present a sentence that says the intention “Intention of Robert-e-ford: go-to-store.”

For project 3 the task was to take the HTN concept and use it to make the scripts an embeddable and recursive hierarchy of scripts. For the implementation I decided to keep the same context free grammar, but that I would look for a few extra parameters in an entry in a script. The extra parameters say whether the entry is part of a sub-script, and also if the entry points to a sub-script: the hierarchy is built out of entries in a script file where some entries have the filename of another script file.

When a script is read, the subscripts it points to are collected and mapped to the script. Next, all subscripts are populated inside the Frame objects that they belong to. At this point, an object hierarchy is created which represents the text files read from disk.

The code that does the pattern matching and printing of the matched event frames now needs to handle this recursive structure. Rather than employing a set of recursive function calls, I chose to keep the same iterative solution because the code that did the statistical counting and pattern matching has gotten rather difficult to read and understand. So

I manually build a list of script entries to process, and it is very much as if I were manually building the stack rather than relying on recursion. The list that is built is really a collapsed version of the hierarchy of scripts. There is a lot of reordering and pruning of items to avoid redundant printouts of matching events. Every event that starts a sub-script is noted so when the printout is done, you don’t get the same result as was present in project 2: when the inputs are run on the aicop program, the resulting outputs are printed out in a hierarchical format (with the exception of a few bugs in indentation of the print outs).

*One nice thing you may notice is that I got things to the point that I can embed the same subscript inside different scripts, and based on the other matches found for entries in the other scripts I match the correct actors. For example the “run-away” script is found in two separate top level scripts.*

*Note: Every sub-script has an intention just as the scripts normally had in project 2.*

## Core Components, Design and Methodologies.

Everything relating to these areas is mostly the same as what is noted in Project2_Report.pdf. But here is a list of changes in each areas:

**Core Components:** There are no new core components that were added to project 3.

**Design:** The overall design was also unaffected.

**Methodologies:** I chose to add some extra fields to my script entry format. If present, the fields have the filenames of other script files. The filenames are collected, and on a separate pass read in. Information about the script name that the sub-script was referenced from is stored so that the sub-script can be added as a sub-script of the correct script frame object in memory.

Finally, a collapsed version of the hierarchy is constructed on the fly so that the code that already worked on the event entries from project 2 is upset as little as possible.

One other methodology I added recently to avoid some issues that I have had (and probably had in Project 2) with collisions of the same strings. For example the causal chain for robbers and the causal chain for robert-e-ford both have an event with action “run-away.” A methodology I do now is to compare the agent for a given event, and if the agent does not match the most frequently observed agent of the chain then I try to find an event with the same action but with an agent that matches the most frequent agent.

Finally another methodology I added was to add a Boolean field to my frame class to tell if it is an intention frame. This way, several frames in various orders in the collapsed list can hold intention information, and I have a special function for printing frames that hold intention data now.

## Self Critique and Evaluation:

The additional changes required to make the solution recursive, and the overall goal of the project were not too ambitious for project 3. So overall the existing code worked fine to achieve the ends of project 3. I would still have the same rantings that I had in project 2 since very little was changed in terms of fixing problems or bugs that I noticed in project2.

One thing I was happy with was the change I mentioned in the methodologies section regarding handling events that match a pattern from the script, but are not in fact the correct event (I call these collisions). As a result of this fix, another thing I was happy with was that I can reuse a script inside several other scripts and get the correct matches.

Overall I still don’t like my implementation for aicop. I think some of it is a result of not knowing the overall goals of aicop from the beginning, and also having originally written it in LISP and then having rewritten it in C++ but with the same design and global variables. It still feels really hacked, but then again this is really my first AI program.

If there is one thing that I could change about the design, it would be to have a more structured set of classes to help in calculating statistics on the frequency of actors used in a given causal chain. I think this would be the most useful improvement because the code that does this right now is all over the place, and very difficult to understand (and also a reason I didn’t implement HTNs using recursive function calls).

One thing to note is that there are some bugs that I have not fixed yet, and they are pretty much all purely cosmetic. I intended that every sub-script print out with a string at the top that says “Sub Script:” and being indented over by one tab. But instead sometimes the tabbing or the “Sub Script:” heading does not print. I didn’t figure out why yet. But all the correct data and intentions for all scripts are printed out in the correct order. The problems with indentation and heading usually happen if too many scripts are embedded 2 or 3 levels deep, **but the correct data and intentions prints in the correct order.**

Otherwise, I am someone impressed with the fact that code that I wrote that is so difficult to read and understand sometimes can seem smart about how it processes certain inputs.

But one last thing. So far, I really rely on the input files being in a correct format. Syntax and semantic type errors in the entries of a script a lot of times show them selves as segfaults.

## Knowledge Representation and Output:

A project 2 script entry would look like the following:

```
(ACPLAN(research-fraud-plan4 research-fraud-plan5
research-fraud-plan)  (investigating      (BLANK           ,
breach-bioethics-laws    ,        , , ,)))
```

Now there are a few new fields that can be put into the set of values inside the parenthesis after ACPLAN. One type of field tells if the entry is part of a sub-script, this is to avoid mixing up sub-scripts with top-level scripts (The way my aicop works is it starts from the top level scripts and processes everything downward, so it will eventually get to the subscripts and I don’t want to doubly print out causal chains).  The other type of field says that the value after it is the name of the text file that stores sub-script information.

An example of the former is the following:

```
(ACPLAN(rob-store-fraud-plan4 rob-store-fraud-plan5
rob-store-fraud-plan ISSUBPLAN TRUE)  (suspect
(BLANK          , fraud      ,        , , ,)))
```

The text “ISSUBPLAN TRUE” tells aicop to not include “rob-store-fraud-plan” in the same collection containing the top level scripts like  “rob-store-plan.”

An example of the other type of field is in the following entry:

```
(ACPLAN(rob-store-fraud-plan3 rob-store-fraud-plan4
rob-store-fraud-plan SUBPLAN fake-id-subplan)  (SUBFOO2
( , , , , ,)))
```

This tells aicop that there is a sub-script in the file “fake-id-subplan.txt” and that file is open and read in by the context free parser on a separate pass of all the subscript filenames collected.

**And now time for an example:**

Here is a top level script for aicop, rob-store-plan.txt:

```
(ACPLAN(rob-store-planACPLAN-START rob-store-plan10 rob-store-plan
SUBPLAN rob-store-fraud-subplan)  (SUBFOO1            ( , , , , ,)))
(ACPLAN(rob-store-plan10 rob-store-planACPLAN-END  rob-store-plan)
(get-rich (agent         ,           ,  , , ,)))
```

It says that the overall goal of the script is to “get-rich” and that the first entry in the script points to another script. So rob-store-fraud-subplan.txt is read in:

```
(ACPLAN(rob-store-fraud-planACPLAN-START rob-store-fraud-plan3 rob-
store-fraud-plan SUBPLAN goto-store-subplan)  (SUBFOO1            ( , ,
, , ,)))
(ACPLAN(rob-store-fraud-plan3 rob-store-fraud-plan4          rob-store-
fraud-plan SUBPLAN fake-id-subplan)  (SUBFOO2            ( , , , , ,)))
(ACPLAN(rob-store-fraud-plan4 rob-store-fraud-plan5          rob-store-
fraud-plan ISSUBPLAN TRUE)  (suspect            (BLANK        , fraud
,        , , ,)))
(ACPLAN(rob-store-fraud-plan5 rob-store-fraud-plan6          rob-store-
fraud-plan ISSUBPLAN TRUE)  (get-caught          (BLANK        ,
,        , , ,)))
(ACPLAN(rob-store-fraud-plan6 rob-store-fraud-plan7          rob-store-
fraud-plan ISSUBPLAN TRUE)  (call               (BLANK        , police    ,
, , ,)))
(ACPLAN(rob-store-fraud-plan7 rob-store-fraud-plan10          rob-store-
fraud-plan SUBPLAN runaway-subplan)  (SUBFOO3            ( , , , ,
,)))
(ACPLAN(rob-store-fraud-plan10 rob-store-fraud-planACPLAN-END  rob-
store-fraud-plan ISSUBPLAN TRUE)  (commit-fraud (agent         ,
,  , , ,)))
```

The first 2 entries point to other sub-scripts, goto-store-subplan.txt and fake-id-subplan.txt, the rest are just normal entries. I could do on, but it should be pretty clear what it is doing at this point.

The output from this script should look something like this:

```
Another Chain: rob-store-plan

Sub Script:

        Sub Script:
               (     ( go-to, ( robert-e-ford,store,,,, )))
               Intention of robert-e-ford:
                      go-to-store


        Sub Script:
               (     ( give, ( robert-e-ford,fake-check,clerk,,, )))
               (     ( give, ( robert-e-ford,fake-id,clerk,,, )))
               Intention of robert-e-ford:
                      fake-identity

        (     ( suspect, ( clerk,fraud,,,, )))
        (     ( get-caught, ( robert-e-ford,,,,, )))
        (     ( call, ( clerk,police,,,, )))

        Sub Script:

                        Sub Script:
                               (     ( run-away, ( robert-e-ford,,,,, )))
                               Intention of robert-e-ford:
                                      runaway

                (     ( chase-down, ( police-officers,robert-e-ford,,,, )))
                (     ( get-away, ( robert-e-ford,accomplice,,,, )))
                Intention of robert-e-ford:
                       run-away-from-cops

        Intention of robert-e-ford:
               commit-fraud

Intention of robert-e-ford:
       get-rich
```

In actuality it looks more like this because I haven’t gotten indentation quite right yet:

```
Another Chain: rob-store-plan

Sub Script:
        (     ( go-to, ( robert-e-ford,store,,,, )))
        Intention of robert-e-ford:
                go-to-store


Sub Script:
        (     ( give, ( robert-e-ford,fake-check,clerk,,, )))
        (     ( give, ( robert-e-ford,fake-id,clerk,,, )))
        Intention of robert-e-ford:
                fake-identity

(     ( suspect, ( clerk,fraud,,,, )))
(     ( get-caught, ( robert-e-ford,,,,, )))
(     ( call, ( clerk,police,,,, )))

Sub Script:
        (     ( run-away, ( robert-e-ford,,,,, )))
        Intention of robert-e-ford:
                runaway

(     ( chase-down, ( police-officers,robert-e-ford,,,, )))
(     ( get-away, ( robert-e-ford,accomplice,,,, )))
Intention of robert-e-ford:
        run-away-from-cops

Intention of robert-e-ford:
        commit-fraud

Intention of robert-e-ford:
        get-rich
```