CS 5600/6600: F24: Intelligent Systems
Assignment 12
Knowledge Engineering for STRIPS

Vladimir Kulyukin
Department of Computer Science
Utah State University

November 16, 2024

Learning Objectives

1. Automated Theorem Proving
2. **ST**anford **R**esearch **I**nstitute **P**roblem **S**olver (STRIPS)
3. Knowledge Engineering STRIPS operators

Problem 1 (2 points)

Read the article "STRIPS: A New Approach to the Application of.Theorem Proving to Problem Solving" by R. Fikes and N. J. Nilsson. The PDF is in the zip. The first version of STRIPS appeared about a decade later after the first version of GPS and, in my opinion, was even more influential on AI planning than GPS. It was, in many respects, more generic than GPS, because it proposed to use automated theorem proving in reasoning about the world.

The first generalization introduced by STRIPS was the uniform adoption of First-Order Predicate Calculus (FOPC) as the knowledge representation formalism. While STRIPS drew heavily on the GPS concept of planning operator, it generalized it to include filters, i.e., statements that must be true for the operator to be considered for potential application. The STRIPS operator preconditions are not only well-formed formulas (wffs) in the planner's knowledge base, but also the wffs that can be inferred by a theorem prover from that base.

Given the autmated theorem proving material we covered in the previous three lectures (FOPC, wffs, unification, substitution, satisfiability, resolution), you should be able to appreciate this article, its main points, and its conclusions. You may want to glance over these PDFs in Canvas and/or your class notes before moving on.

Solving Sussman Anomaly with STRIPS

The directory `strips` in the zip contains the source code of a STRIPS implementation. Runing STRIPS is very similar to running GPS. You load the system and then load the operators for a specific world (i.e., universe of discourse). The file `loader.lisp` has the function `load-strips` that loads all the files into Lisp. The variable `*strips-files*` contains the list of all files that must be loaded for STRIPS to run. Let's load it.

```
> (load "loader.lisp")
;; Loading file loader.lisp ...
```

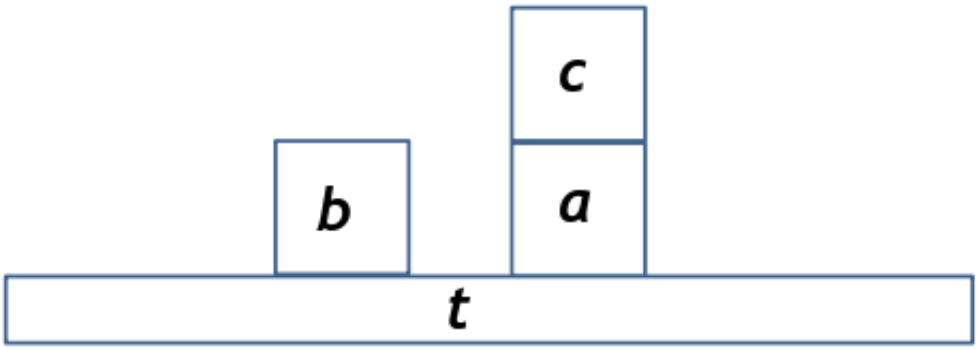

Figure 1: Sussman Anomaly in the blocks world.

```
;; Loaded file loader.lisp
#P"/home/vladimir/teaching/AI/F24/hw/hw_strips/strips/loader.lisp"
> (load-strips)
...
;; Loaded file strips.lisp
T
```

A STRIPS programmer must knoledge enginner two files for STRIPS to function properly in a specific possible world: a file with wffs and a file with operators. Let's take a look at two such files for the Sussman Anomaly (cf. Figure 1) in the `strips` directory: `sa.wff` and `sa.op`. The `.wff` file contains an FOPC formalization of the initial state world. The `.op` contains the operators for the plant to operate in the world.

If you open `sa.wff`, you'll see a bunch of store commands. The function `store` is a tool that stores FOPC wffs in the STRIPS knowledge base (aka knowledge database).

```
(store '(inst A block))
(store '(inst B block))
(store '(inst C block))
(store '(inst T table))
(store '(clear C))
(store '(clear B))
(store '(on B T))
(store '(on A T))
(store '(on C A))
```

The conceptualization for this world formalized with FOPC includes 4 objects (i.e., the blocks `a`, `b`, `c`, and table `t`), no functions, and the relations `inst`, `clear`, and `on`. The wff `(inst A block)` states that the object corresponding to the constant `A` (under some interpretation function) is an instance of the concept `block`.

Let's load these wffs.

```
> (load "sa.wff")
```

Automated Theorem Provers vs. Inference Engines

We can now play with our simple theorem prover implemented in the function `show` in `show.lisp`. It is not nearly as complicated or complete as QA3.5, the theorem prover used in the system presented in Fikes and Nilsson's paper, but it is much faster.

Note in this regard the following statement made by Fikes and Nilsson in their paper: "We note that in all cases most of the time is spent doing theorem proving (in QA3.5)." Theorem proving is a slow process even on modern CPU/GPU architectures. There are many possible proof paths to follow. Some researchers argue that quantum computers will speed things up for us and quantum theorem provers will put mathematicians out of business. Well, we'll see.

Symbolic AI programmers often sacrifice completeness for speed by implementing what is known as an *inference engines*. An inference engine is an incomplete theorem prover in that there are wffs implied by the base that it won't be able to derive mechanically. However, those wffs that it can prove, it can prove in a reasonable amount of time.

Let's play with our inference engine in `show.lisp`.

```
> (show '(inst A block))
((NIL))
```

The above call asks the engine to show that `A` is a block. The engine succeeds, because `((NIL))` is returned. The empy list inside the returned list, i.e., `(NIL)` means that no variable bindings were found during the unification process, because the wff is in the base. The engine returns the same result when asked to prove that `B` and `C` are blocks.

```
> (show '(inst B block))
((NIL))
> (show '(inst C block))
((NIL))
```

If we ask the engine to prove that `T` is a block, it returns `NIL`, i.e., it cannot be satisifed.

```
> (show '(inst T block))
NIL
```

To reiterate, `((NIL))` means that the wff is satisfied, i.e., the engine can derive it from the database, but no bindings were found during unification. If, however, the engine returns `NIL`, it means that the wff cannot be found satisfiable by the engine.

Caveat: the latter result does not mean that the wff is *really* not satisfiable, because a more complete engine or a human theorem prover may be able to find it satisfiable from the base. Let's ask the engine if `T` is a table. Remember that once we choose to use inference engines, we've sacrifisied completeness for speed.

```
> (show '(inst T table))
((NIL))
```

Yes, `(inst T table)` is satisfiable. Let's ask the engine to prove that there exists a block in the world. To be more exact, we are asking the engine to prove that the existence of blocks is satisfiable in the current formalization of the world. Meditate on the previous sentence for a few moments. It's worth it!

We ask the engine to satisfy `(inst ?w block)`. Recall that `?w` is a variable. The engine uses unification to see if there are wffs in its base that can be unified with this wff, and, if yes, returns the bindings.

```
> (show '(inst ?w block))
(((W C)) ((W B)) ((W A)))
```

The returned list of bindings means that `?w` can be bound to `C`, `B`, and `A`. And, indeed, all these constants can be interpreted (under a specific interpretation function) as block objects in our universe of discourse. Let's ask the engine if there are tables in our world.

```
> (show '(inst ?z table))
(((Z T)))
```

From the list of bindings returned by the engine we can tell that there is only table object to which our interpretation function maps the constant `T`. Again, medidate on the meaning of the previous sentence for a few moments.

```
> (show '(inst ?z table))
(((Z T)))
> (show '(inst ?z monkey))
NIL
```

Alas, there are no constants that map to monkeys in this world, which makes it somewhat boring. We'll fix it in the next problem. But, let's push on and ask if `B` and `A` are clear to see that `B` is and `A` is not.

```
> (show '(clear B))
((NIL))
> (show '(clear A))
NIL
```

A more general form of this question is to ask the engine if there are clear objects in the world. To put it differently, we ask the engine to prove a theorem that there are blocks stacked on each other. And, yes, there are such blocks.

```
> (show '(and (on ?w ?z) (inst ?w block) (inst ?z block)))
((NIL (Z A) (W C)))
```

However, there are no objects `?w` and `?z` such that `?w` is on `?z` and `?z` is clear.

```
(show '(and (on ?w ?z) (clear ?z)))
NIL
```

STRIPS Operators

Let's now talk about STRIPS operators. STRIPS borrowed heavily from GPS as far as operators go, but there are a few important differences. In STRIPS, an operator has the following structure.

```
(def-operator <op-name>
  :action     <action>
  :goal       <goal>
  :precond    <preconditions>
  :filter     <filter>
  :add        <list of wffs to add>
  :del        <list of wffs to delete>)
```

The filter is a list wffs that must be satisifed in the current state of the world for the operator to be applicable. The difference between a precondition and a filter is that a precondition can become a goal for which other operators must be found whereas a filter must be directly satisified in the current state of the world.

Let's take a look at the operators in `sa.op`. We will make the plant's operation in this world as abstract as possible by having only one operator type – Schank's `PTRANS`. Recall from our conceptual dependency theory lectures and Dunlop's paper, that `PTRANS` denotes any physical transfer, i..e, the transfer of the physical location in some coordinate system of some object.

We will use only filter wffs in our operators. Do we have to do it? No! But it's simpler and, hence, faster that way. Open up `sa.op` and let's look at the first `PTRANS` operator.

```
(def-operator PTRANS
  :action (PTRANS plant ?b1 ?b2 ?t)
  :goal   (on ?b1 ?t)
  :precond ()
  :filter ((on ?b1 ?b2)
           (inst ?b1 block)
           (inst ?b2 block)
           (inst ?t  table))
  :add ((on ?b1 ?t) (clear ?b2))
  :del ((on ?b1 ?b2)))
```

The operator signature in `:action` is `(PTRANS plant <object> <from> <to>)`, where the plant is some virtual or real platform for which STRIPS is planning (e.g., an office cleaning robot), and the other arguments are variables. In the CD representation we studied in CA and SAM, this is equivalent to

```
(PTRANS :ACTOR  (PLANT)
        :OBJECT ?OBJ
        :FROM   ?FROM
        :TO     ?TO).
```

The goal is `(on ?b1 ?t)`. In other words, in the state of the world after the application of the operator `(on ?b1 ?t)` must be satisfiable. If you look at the filters, it's not just any constant that can bind to `?b1` and `?t`. The constant that binds to `?b1` must be a block and the constant that binds to `?t` must be a table. The filters also state that this operator is applicable if and only if `?b1`, `?b2` are blocks `?t` is a table and the binding of `?b1` is on the binding of `?b2`.

The add list includes the wffs `(on ?b1 ?t)` and `(clear ?b2)`. The delete list includes the wff `(on ?b1 ?b2)`, which means that `?b1` is no longer on `?b2`. The semantics of this operator is that the plant moves the block `?b1` from the block `?b2` to the table `?t`.

Here's the second PTRANS operator. This one allows the plant to PTRANS the block `?b` from the table `?t` to the block `?b2` on the assumption that the filters are satisfied.

```
(def-operator PTRANS
  :action (PTRANS plant ?b1 ?t ?b2)
  :goal   (on ?b1 ?b2)
  :precond ()
  :filter ((on   ?b1 ?t)
           (inst ?b1 block)
           (inst ?t  table)
```

```
          (clear ?b1)
          (inst ?b2 block))
  :add ((on ?b1 ?b2))
  :del ((clear ?b2) (on ?b1 ?t)))
```

The third PTRANS operator is for the plant to move the block `?b1` from the block `?b2` on to the block `?b3`. Here it is.

```
(def-operator PTRANS
  :action (PTRANS plant ?b1 ?b2 ?b3)
  :goal   (on ?b1 ?b3)
  :precond ()
  :filter ((on   ?b1 ?b2)
           (inst ?b1 block)
           (inst ?b2 block)
           (inst ?b3 block)
           (clear ?b1)
   (clear ?b3))
  :add ((on ?b1 ?b3))
  :del ((clear ?b3) (on ?b1 ?b2)))
```

Finally, we define a kickstarter operator that allows us to start the planning process by formulating the end goal that our collection of operators will (we hope!) satisfy.

```
(def-operator DONE
  :action     (done)
  :goal       (sussman-solved)
  :precond    ((on C T) (on B C) (on A B))
  :add        ((sussman-solved))
  :del        ())
```

Let's load and run.

```
> (load "sa.op")
;; Loading file sa.op ...
;; Loaded file sa.op
#P"/home/vladimir/teaching/AI/F24/hw/hw_strips/strips/sa.op"
> (strips '(sussman-solved))

 (GOAL (SUSSMAN-SOLVED))
...

PLAN:
 (PTRANS PLANT C A T)
 (PTRANS PLANT B T C)
 (PTRANS PLANT A T B)
 (DONE)

((:OPERATOR DONE :ACTION (DONE)
  :GOAL (SUSSMAN-SOLVED)
  :PRECOND ((ON C T) (ON B C) (ON A B))
  :FILTER NIL
  :ADD ((SUSSMAN-SOLVED))
```

```
  :DEL NIL)
 (:OPERATOR PTRANS
  :ACTION (PTRANS PLANT A T B)
  :GOAL (ON A B)
  :PRECOND NIL
  :FILTER ((ON A T) (INST A BLOCK) (INST T TABLE) (CLEAR A) (INST B BLOCK))
  :ADD ((ON A B))
  :DEL ((CLEAR B) (ON A T)))
 (:OPERATOR PTRANS
  :ACTION (PTRANS PLANT B T C)
  :GOAL (ON B C)
  :PRECOND NIL
  :FILTER ((ON B T) (INST B BLOCK) (INST T TABLE) (CLEAR B) (INST C BLOCK))
  :ADD ((ON B C))
  :DEL ((CLEAR C) (ON B T)))
 (:OPERATOR PTRANS
  :ACTION (PTRANS PLANT C A T)
  :GOAL (ON C T) :PRECOND NIL
  :FILTER ((ON C A) (INST C BLOCK) (INST A BLOCK) (INST T TABLE))
  :ADD ((ON C T) (CLEAR A))
  :DEL ((ON C A))))
```

If you examine the above output (I omitted a few things and indented a few things for legibility) of STRIPS, it consists of two logical components. The found plan printed out as a sequence of actions, i.e.,

```
PLAN:
 (PTRANS PLANT C A T)
 (PTRANS PLANT B T C)
 (PTRANS PLANT A T B)
 (DONE)
```

and the actual sequence of instantiated operators, in reverse order, that the planner has found.

```
((:OPERATOR DONE :ACTION (DONE)
  :GOAL (SUSSMAN-SOLVED)
  :PRECOND ((ON C T) (ON B C) (ON A B))
  :FILTER NIL
  :ADD ((SUSSMAN-SOLVED))
  :DEL NIL)
 (:OPERATOR PTRANS
  :ACTION (PTRANS PLANT A T B)
  :GOAL (ON A B)
  :PRECOND NIL
  :FILTER ((ON A T) (INST A BLOCK) (INST T TABLE)
           (CLEAR A) (INST B BLOCK))
  :ADD ((ON A B))
  :DEL ((CLEAR B) (ON A T)))
 (:OPERATOR PTRANS
  :ACTION (PTRANS PLANT B T C)
  :GOAL (ON B C)
  :PRECOND NIL
  :FILTER ((ON B T) (INST B BLOCK) (INST T TABLE)
```

```
          (CLEAR B) (INST C BLOCK))
  :ADD ((ON B C))
  :DEL ((CLEAR C) (ON B T)))
 (:OPERATOR PTRANS
  :ACTION (PTRANS PLANT C A T)
  :GOAL (ON C T) :PRECOND NIL
  :FILTER ((ON C A) (INST C BLOCK) (INST A BLOCK)
           (INST T TABLE))
  :ADD ((ON C T) (CLEAR A))
  :DEL ((ON C A))))
```

If we want to actually use the constructed plan, i.e., the sequence of operators, or feed it to another symbolic AI system, i.e., CA to generate a verbal description of the plan to a human user or to SAM to scriptify the plan and file it away for the future, here's a way to do it.

```
> (setf sussman-plan (reverse (strips '(sussman-solved))))
> sussman-plan
((:OPERATOR PTRANS :ACTION (PTRANS PLANT C A T) :GOAL (ON C T) :PRECOND NIL
  :FILTER ((ON C A) (INST C BLOCK) (INST A BLOCK) (INST T TABLE)) :ADD
  ((ON C T) (CLEAR A)) :DEL ((ON C A)))
 (:OPERATOR PTRANS :ACTION (PTRANS PLANT B T C) :GOAL (ON B C) :PRECOND NIL
  :FILTER ((ON B T) (INST B BLOCK) (INST T TABLE) (CLEAR B) (INST C BLOCK))
  :ADD ((ON B C)) :DEL ((CLEAR C) (ON B T)))
 (:OPERATOR PTRANS :ACTION (PTRANS PLANT A T B) :GOAL (ON A B) :PRECOND NIL
  :FILTER ((ON A T) (INST A BLOCK) (INST T TABLE) (CLEAR A) (INST B BLOCK))
  :ADD ((ON A B)) :DEL ((CLEAR B) (ON A T)))
 (:OPERATOR DONE :ACTION (DONE) :GOAL (SUSSMAN-SOLVED) :PRECOND
  ((ON C T) (ON B C) (ON A B)) :FILTER NIL :ADD ((SUSSMAN-SOLVED)) :DEL NIL))
```

Problem 2 (3 points): Monkey and Banana in STRIPS

Let's solve the monkey and banana problem in STRIPS. The wffs for this world are in `mb.wff`

```
(store '(inst CHAIR1    chair))
(store '(inst MONKEY1   monkey))
(store '(inst ROOM1     room))
(store '(inst BANANA1   banana))
(store '(inst MID-ROOM1 mid-room))
(store '(inst DOOR1     door))
(store '(inst FLOOR1    floor))
(store '(inst BALL1     ball))
(store '(at MONKEY1     MID-ROOM1))
(store '(at CHAIR1      DOOR1))
(store '(on MONKEY1     FLOOR1))
(store '(has MONKEY1    BALL1))
(store '(hungry         MONKEY1))
```

There is a chair CHAIR1, a monkey MONKEY1, a room ROOM1, a banana BANANA1, a mid-room MID-ROOM1, a door DOOR1, a floor FLOOR1, a ball BALL1. MONKEY1 is at MID-ROOM1. CHAIR1 is at DOOR1. MONKEY1 is on FLOOR1. MONKEY1 has BALL1. MONKEY1 is hungry. You can load this wff file and play with the inference engine.

Let's look at `mb.op`. I wrote a couple of operators for you as examples and left a few operators for you to write to experience STRIPS better. If you solved this problem in GPS, you have a good grounding in this problem. Here's one such operator.

```
(def-operator PTRANS
  :action (PTRANS ?monkey ?monkey ?mid-room ?door)
  :goal (at ?monkey ?door)
  :precond ((at ?monkey ?mid-room)
            (on ?monkey ?floor))
  :filter ((inst ?door   door)
           (inst ?mid-room mid-room)
           (inst ?monkey monkey)
           (inst ?floor  floor))
  :add ((at ?monkey ?door))
  :del ((at ?monkey ?mid-room)))
```

In CD, this operator represents the following action.

```
(PTRANS (ACTOR ?MONKEY) (OBJECT ?MONKEY) (FROM ?MID-ROOM) (TO ?DOOR))
```

To satisfy the goal `(at ?monkey ?door)`, the monkey must be in the middle of the room and on the floor. If these two preconditions are satisfied, the monkey can PTRANS himself from the middle of the room to the door.

Here's the plan that STRIPS finds with my operators.

```
> (load "mb.wff")
;; Loaded file mb.wff
#P"/home/vladimir/teaching/AI/F24/hw/hw_strips/strips/mb.wff"
> (load "mb.op")
;; Loading file mb.op ...
;; Loaded file mb.op
#P"/home/vladimir/teaching/AI/F24/hw/hw_strips/strips/mb.op"
> (strips '(nourished MONKEY1))

 (GOAL (NOURISHED MONKEY1))
...
PLAN:
 (PTRANS MONKEY1 MONKEY1 MID-ROOM1 DOOR1)
 (PTRANS MONKEY1 BALL1 MONKEY1 FLOOR1)
 (PTRANS MONKEY1 CHAIR1 DOOR1 MID-ROOM1)
 (PTRANS MONKEY1 MONKEY1 FLOOR1 CHAIR1)
 (GRASP MONKEY1 BANANA1)
 (INGEST MONKEY1 BANANA1)

((:OPERATOR INGEST :ACTION (INGEST MONKEY1 BANANA1) :GOAL (NOURISHED MONKEY1)
  :PRECOND ((HUNGRY MONKEY1) (HAS MONKEY1 BANANA1)) :FILTER
  ((INST MONKEY1 MONKEY) (INST BANANA1 BANANA)) :ADD
  ((EMPTY-HANDED MONKEY1) (NOURISHED MONKEY1)) :DEL
  ((HAS MONKEY1 BANANA1) (HUNGRY MONKEY1)))
 (:OPERATOR GRASP :ACTION (GRASP MONKEY1 BANANA1) :GOAL (HAS MONKEY1 BANANA1)
  :PRECOND ((AT CHAIR1 MID-ROOM1) (ON MONKEY1 CHAIR1) (EMPTY-HANDED MONKEY1))
  :FILTER
  ((INST MONKEY1 MONKEY) (INST CHAIR1 CHAIR) (INST MID-ROOM1 MID-ROOM)
   (INST BANANA1 BANANA))
  :ADD ((HAS MONKEY1 BANANA1)) :DEL ((EMPTY-HANDED MONKEY1)))
 (:OPERATOR PTRANS :ACTION (PTRANS MONKEY1 MONKEY1 FLOOR1 CHAIR1) :GOAL
  (ON MONKEY1 CHAIR1) :PRECOND
```

```
  ((AT CHAIR1 MID-ROOM1) (AT MONKEY1 MID-ROOM1) (ON MONKEY1 FLOOR1)) :FILTER
  ((INST CHAIR1 CHAIR) (INST MID-ROOM1 MID-ROOM) (INST MONKEY1 MONKEY)
   (INST FLOOR1 FLOOR))
  :ADD ((ON MONKEY1 CHAIR1) (AT MONKEY1 ?BANANA)) :DEL
  ((AT MONKEY1 MID-ROOM1) (ON MONKEY1 FLOOR1)))
 (:OPERATOR PTRANS :ACTION (PTRANS MONKEY1 CHAIR1 DOOR1 MID-ROOM1) :GOAL
  (AT CHAIR1 MID-ROOM1) :PRECOND
  ((AT CHAIR1 DOOR1) (AT MONKEY1 DOOR1) (ON MONKEY1 FLOOR1)
   (EMPTY-HANDED MONKEY1))
  :FILTER
  ((INST CHAIR1 CHAIR) (INST DOOR1 DOOR) (INST MONKEY1 MONKEY)
   (INST FLOOR1 FLOOR))
  :ADD ((AT MONKEY1 MID-ROOM1) (AT CHAIR1 MID-ROOM1)) :DEL
  ((AT MONKEY1 DOOR1) (AT CHAIR1 DOOR1)))
 (:OPERATOR PTRANS :ACTION (PTRANS MONKEY1 BALL1 MONKEY1 FLOOR1) :GOAL
  (EMPTY-HANDED MONKEY1) :PRECOND ((HAS MONKEY1 BALL1)) :FILTER
  ((INST MONKEY1 MONKEY) (INST BALL1 BALL) (INST FLOOR1 FLOOR)) :ADD
  ((EMPTY-HANDED MONKEY1) (ON BALL1 ?LOOR)) :DEL ((HAS MONKEY1 BALL1)))
 (:OPERATOR PTRANS :ACTION (PTRANS MONKEY1 MONKEY1 MID-ROOM1 DOOR1) :GOAL
  (AT MONKEY1 DOOR1) :PRECOND ((AT MONKEY1 MID-ROOM1) (ON MONKEY1 FLOOR1))
  :FILTER
  ((INST DOOR1 DOOR) (INST MID-ROOM1 MID-ROOM) (INST MONKEY1 MONKEY)
   (INST FLOOR1 FLOOR))
  :ADD ((AT MONKEY1 DOOR1)) :DEL ((AT MONKEY1 MID-ROOM1))))
```

Here's the plan in the actual order of the found operators.

```
> (setf mb-plan (reverse (strips '(nourished MONKEY1))))
...
> mb-plan
((:OPERATOR PTRANS :ACTION (PTRANS MONKEY1 MONKEY1 MID-ROOM1 DOOR1)
  :GOAL (AT MONKEY1 DOOR1)
  :PRECOND ((AT MONKEY1 MID-ROOM1) (ON MONKEY1 FLOOR1))
  :FILTER
  ((INST DOOR1 DOOR) (INST MID-ROOM1 MID-ROOM) (INST MONKEY1 MONKEY)
   (INST FLOOR1 FLOOR))
  :ADD ((AT MONKEY1 DOOR1))
  :DEL ((AT MONKEY1 MID-ROOM1)))
 (:OPERATOR PTRANS
  :ACTION (PTRANS MONKEY1 BALL1 MONKEY1 FLOOR1)
  :GOAL (EMPTY-HANDED MONKEY1)
  :PRECOND ((HAS MONKEY1 BALL1))
  :FILTER ((INST MONKEY1 MONKEY) (INST BALL1 BALL) (INST FLOOR1 FLOOR))
  :ADD ((EMPTY-HANDED MONKEY1) (ON BALL1 ?LOOR))
  :DEL ((HAS MONKEY1 BALL1)))
 (:OPERATOR PTRANS
  :ACTION (PTRANS MONKEY1 CHAIR1 DOOR1 MID-ROOM1)
  :GOAL (AT CHAIR1 MID-ROOM1)
  :PRECOND
  ((AT CHAIR1 DOOR1) (AT MONKEY1 DOOR1) (ON MONKEY1 FLOOR1)
   (EMPTY-HANDED MONKEY1))
  :FILTER
  ((INST CHAIR1 CHAIR) (INST DOOR1 DOOR) (INST MONKEY1 MONKEY)
```

```
  (INST FLOOR1 FLOOR))
 :ADD ((AT MONKEY1 MID-ROOM1) (AT CHAIR1 MID-ROOM1))
 :DEL ((AT MONKEY1 DOOR1) (AT CHAIR1 DOOR1)))
(:OPERATOR PTRANS
 :ACTION (PTRANS MONKEY1 MONKEY1 FLOOR1 CHAIR1)
 :GOAL (ON MONKEY1 CHAIR1)
 :PRECOND ((AT CHAIR1 MID-ROOM1) (AT MONKEY1 MID-ROOM1) (ON MONKEY1 FLOOR1))
 :FILTER
 ((INST CHAIR1 CHAIR) (INST MID-ROOM1 MID-ROOM) (INST MONKEY1 MONKEY)
  (INST FLOOR1 FLOOR))
 :ADD ((ON MONKEY1 CHAIR1) (AT MONKEY1 ?BANANA))
 :DEL ((AT MONKEY1 MID-ROOM1) (ON MONKEY1 FLOOR1)))
(:OPERATOR GRASP
 :ACTION (GRASP MONKEY1 BANANA1)
 :GOAL (HAS MONKEY1 BANANA1)
 :PRECOND ((AT CHAIR1 MID-ROOM1) (ON MONKEY1 CHAIR1) (EMPTY-HANDED MONKEY1))
 :FILTER
 ((INST MONKEY1 MONKEY) (INST CHAIR1 CHAIR) (INST MID-ROOM1 MID-ROOM)
  (INST BANANA1 BANANA))
 :ADD ((HAS MONKEY1 BANANA1))
 :DEL ((EMPTY-HANDED MONKEY1)))
(:OPERATOR INGEST
 :ACTION (INGEST MONKEY1 BANANA1)
 :GOAL (NOURISHED MONKEY1)
 :PRECOND ((HUNGRY MONKEY1) (HAS MONKEY1 BANANA1))
 :FILTER
 ((INST MONKEY1 MONKEY) (INST BANANA1 BANANA))
 :ADD ((EMPTY-HANDED MONKEY1) (NOURISHED MONKEY1))
 :DEL ((HAS MONKEY1 BANANA1) (HUNGRY MONKEY1))))
```

What to Submit

Save your 1-page paper writeup in `strips_paper.pdf` and your operators in `mb.op` and submit both files in Canvas.

Happy Reading, Writing, and Knowledge Engineering for STRIPS!