MASSACHUSETTS INSTITUTE OF TECHNOLOGY

A.I. LABORATORY

January 1970

Artificial Intelligence
Memo No. 254

LOGO
Memo No. 5

NIM: A Game-Playing Program

Seymour Papert

and

Cynthia Solomon

This work was supported by the National Science Foundation under grant number GJ-1049 and conducted at the Artificial Intelligence Laboratory, a Massachusetts Institute of Technology research program supported in part by the Advanced Research Projects Agency of the Department of Defense and monitored by the Office of Naval Research under Contract Number N00014-70-A-0362-0002.

NIM: A Game-Playing Program

by

Seymour Papert and Cynthia Solomon

1.0 Introduction

This note illustrates some ideas about how to initiate beginning students into the art of planning and writing a program complex enough to be considered a project rather than an exercise on using the language or simple programming ideas. The project is to write a program to play a simple game ("one-pile NIM" or "21") as invincibly as possible. We developed the project for a class of seventh grade children we taught in 1968-69 at the Muzzey Junior High School in Lexington, Mass.* This was the longest programming project these children had encountered, and our intention was to give them a model of how to go about working under these conditions. To achieve this purpose we ourselves worked very hard to develop a clear organization of sub-goals which we explained to the class at the beginning of the 3 - week period devoted to this particular program. One would not expect beginners to find as clear a subgoal structure as this one; but once they have seen a good example, they are more likely to do so in the future for other problems. Thus our primary teaching purpose was to develop the idea of splitting a task into sub-goals. We wanted the children to have good models of various ways in which this can be done and to experience the heuristic power of this kind of planning (as opposed to jumping straight into writing programs).

Readers will notice that the sub-goal structure divides the problem in several ways. One way is by "chopping", that is to say, by recognizing that the final program has distinct functions that can be performed by separate sub-procedures. But this is not the only way. Many heuristic

---

*This work was supported by NSF Contract No. NSF-C 558 to Bolt, Baranek and Newman, Inc.

programs can be "simplified" rather than "chopped". We illustrate this by first writing a procedure to play the "whole game", but in a "dumb way". Once we have done so, we can study its performance, decide why it plays badly and strengthen its play. Thus the successive partial solutions to the problem appear as making a procedure progressively "smarter".

Describing the evolution of the program in this way has the additional benefit of allowing one to make an analogy with the way a child might learn the game. We find this analogy valuable in two senses: by using himself as a model the child acquires a fertile source of ideas about programming; on the other hand, the experience of debugging programs can have a therapeutic effect in leading him to see his own as emotionally neutral bugs rather than as emotionally charged errors.

1.1 The Sub-Goal Plan

The key idea for subdivision of the problem is to write a series of programs, each of which is "smarter" than the previous one. The first program will know nothing about the strategy of play. It will not generate moves, but ask each of two human players in turn what move to make. For example, it might act as a score-keeper, just keeping track of the number of sticks without bothering about whether the move is legal. From score-keeper the machine could advance to referee. This means that it checks the moves for legality and eventually declares the game over and announces the winner. After we have a working mechanical referee we will start making a mechanical player. The first version of a player will choose legal, but not necessarily good moves. Indeed, it will generate a move randomly, use its ability as a referee to decide if it is legal, and then either accept it or generate another random move.

When this works, the child may make his program smarter and smarter by adding features or by writing a completely new version until finally -- if all goes well -- an infallible strategic player is evolved.

A natural form for programs of intermediate "smartness" is the following: the program has a list of simple situations in which it knows how to play; in other situations it plays randomly. In other words, it plays by the form of strategy used by most children in most strategic games.

> In working with a class, a good moment should be seized to prod the children into noting and discussing the analogy between this very simple heuristic program and themselves -- particularly, how the program gets to be "smarter" through more or through better knowledge. Seeing the program as a cognitive model is a valuable and exciting experience for the children. They can easily be drawn into discussion about how meaningful such models are. To keep the discussion alive the teacher should be equipped with arguments and examples to counteract extremist, and so sterile, positions. For example, if the children feel that the program is too simple to be a model of human thinking, one might discuss whether a toy airplane is a useful model of a jet-liner. Does it work by the same principles? Can one learn about airliners by studying toy models? On the other hand, if a class swings over to the position that there really is no difference, one could ask questions about whether the program could learn by itself without a programmer. But if this is too enthusiastically accepted it is well to ask: how much do you learn without being told? Etc., etc. Ideally, the teacher should merely guide the discussion without having to say any of this. But awareness of such arguments will permit more sensitive guiding. An interesting exercise and base for discussion is to have the children study various programs of intermediate smartness, classify their bad moves by degrees of stupidity, give the programs grades or I.Q.'s (or say why they think doing so is silly!).

The stratification of the project has the good feature of allowing children to find their own level. A slower child who gets only as far as the random player, nevertheless, has the taste of success if his program does what it does well. Tendencies to feel inferior should be counteracted by the teacher's attitude and by encouraging individual

variations so that no child's final program is a mere subset of a more advanced one. The teacher's computer culture can be very relevant in this delicate kind of situation. Although the richness of programming permits children to generate many fertile ideas, sensitive filtering by the teacher can enormously improve the achievement-to-frustration ratio.

> Examples of individual frills to a referee program: timing moves, declaring the winner a move or two ahead(!), allowing a player to take a move back, printing a score sheet, giving advice (!), allowing the players to be at two teletypes (if the system permits), establishing and imposing handicaps (!), changing the rules, etc., etc.

2.0 The Rules

A move consists of taking one, two or three match-sticks from a given pile. Two players move alternately. The player who takes the last stick wins.

3.0. First Steps with the Children

The first step is to see that everyone knows the rules and understands what the first program will do; for example, by imitating its function or by writing imaginary scripts. In the course of discussing this we would introduce some names (so as to be able to talk about what we are doing!).

Example of a Script

```
THE NUMBER OF STICKS IS 8
JON TO PLAY.  WHAT'S YOUR MOVE?
<2

THE NUMBER OF STICKS IS 6
BILL TO PLAY.  WHAT'S YOUR MOVE?
<3

THE NUMBER OF STICKS IS 3
JON TO PLAY.  WHAT'S YOUR MOVE?
<3

JON IS THE WINNER.
```

Later in the project we insist that children consider what happens when a player replies to "WHAT'S YOUR MOVE?" by "5" or "COW". In the beginning we would discourage all but the most competent children from worrying about "funny" answers before getting the program to work with normal answers.

Examining the script we see that there must be names for:

the current number of sticks -- let's say "STICKS"

the move -- let's say "MOVE"

the next player -- let's say "PLAYER"

and, a little more subtle

the other player -- let's say "OPPONENT" .

To be sure that everyone understands we have an assignment to fill in these LOGOTHINGS for successive rounds following the previous script.

ROUND #	:STICKS	:PLAYER	:OPPONENT	:MOVE
1	8	"JON"	"BILL"	2
2	_	_	"JON"	3
3	3	_	_	_

4.0. A Simple Score-keeper

If this is the first game-playing program, we might give the class an almost ready-made procedure. We build up to it by asking some standard questions:

What shall we call the procedure? (Let's say "NIMPLAY")

What must NIMPLAY do?

What must NIMPLAY know?

Possible answers are:

1. Announce the remaining number of sticks
2. Announce the player to move
3. Get his move and make all the modifications
4. Recur.

To do this NIMPLAY must remember !STICKS , :PLAYER , and :OPPONENT from the previous round and get :MOVE by asking for it. The first three THINGS must be told by one NIMPLAY-GUY* to another, so they should be inputs. On the other hand, :MOVE comes from the human player, so it can be gotten by REQUEST and need not be an input. If one looks ahead one might notice that later on, :MOVE will sometimes come from a procedure

*The anthropomorphic metaphor is related to the little-men pictures in an earlier section. The use of the anthropomorphic language might be a little precious, but the concept of a separate agent for each program-call is enormously valuable. The children did not seem to resent terms like "MAN" or "GUY".

-- that is, when the machine gets to be smart enough to make its own moves. So to keep the door open for changes, we separate the problems of getting :MOVE and using it. The standard way to do this is to plan on a sub-procedure -- say, called "GETMOVE".

Now we can write NIMPLAY:

```
TO NIMPLAY :STICKS  :PLAYER  :OPPONENT
1 PRINT SENTENCE "THE NUMBER OF STICKS IS" :STICKS
2 PRINT SENTENCE :PLAYER  "TO PLAY.  WHAT'S YOUR MOVE?"
3 MAKE
    NAME   "NEWSTICKS"
    THING   :STICKS  - GETMOVE
4 NIMPLAY :NEWSTICKS  :OPPONENT  :PLAYER

END
```

When in doubt have lots of inputs.

Announce the number of sticks.

We pretend we have already written GETMOVE.

Recursion line. Notice how :PLAYER and :OPPONENT are reversed.

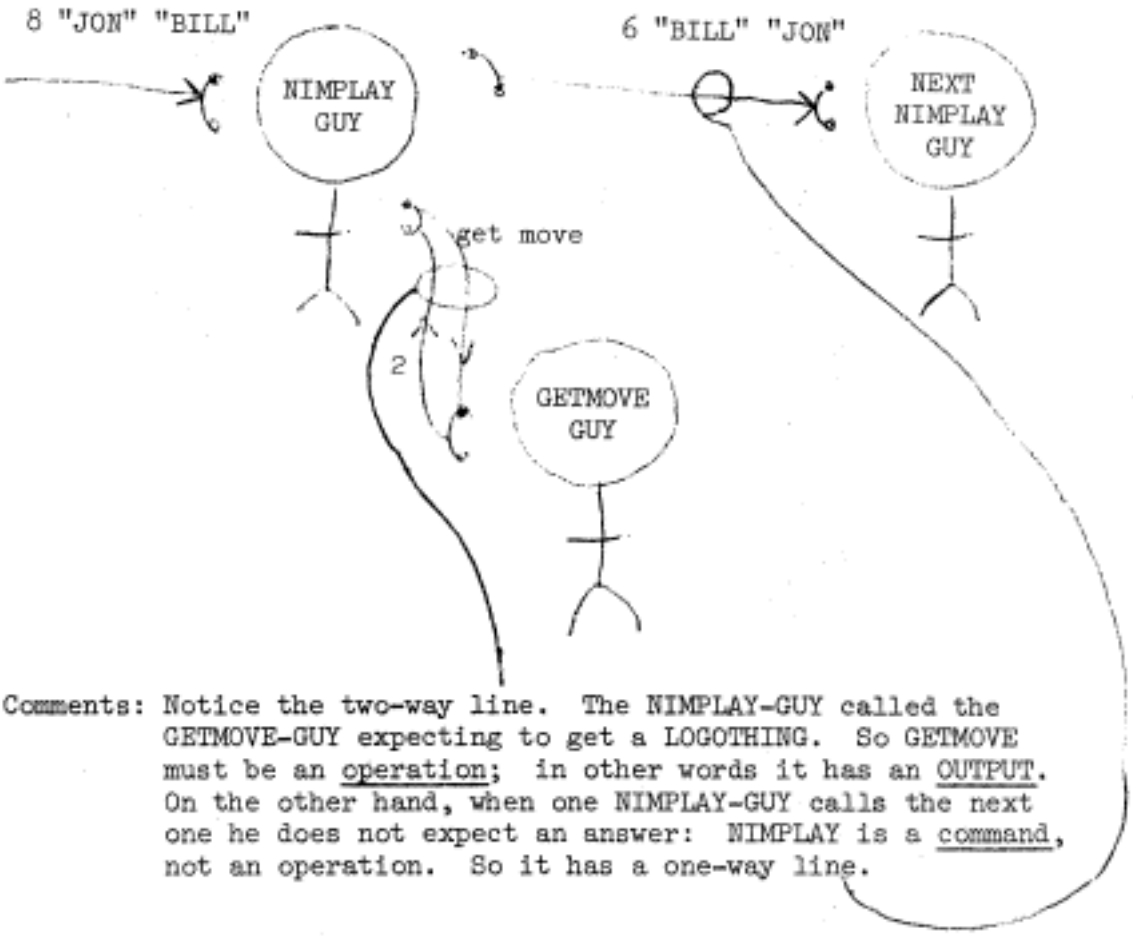
```
TO GETMOVE

1 MAKE
    NAME   "MOVE"
    THING   REQUEST

2 OUTPUT :MOVE

END
```

No input is necessary.

GETMOVE's job is to make a new LOGOTHING. So its main action is this MAKE command. It uses OUTPUT to pass on what it makes.

Note the use of :STICKS -GETMOVE. We use infix notation as an option in LOGO (with parentheses when needed to avoid ambiguity).

A little-man picture of a round:



Comments: Notice the two-way line. The NIMPLAY-GUY called the GETMOVE-GUY expecting to get a LOGOTHING. So GETMOVE must be an operation; in other words it has an OUTPUT. On the other hand, when one NIMPLAY-GUY calls the next one he does not expect an answer: NIMPLAY is a command, not an operation. So it has a one-way line.

5.0 From Score-keeper to Referee

As referee the program has some new tasks:

1. Decide whether the game is over
2. Declare the winner if it is over
3. Make sure that :PLAYER takes 1, 2, or 3 sticks each time.

The first tasks are achieved by adding a STOP-TEST line to NIMPLAY.

For example,

```
TEST IS :NEWSTICKS  ∅
IFTRUE PRINT SENTENCE :PLAYER AND "IS THE WINNER"
IFTRUE STOP
```

The third task can be accomplished by giving GETMOVE a TRY-AGAIN form.

```
TO GETMOVE
1 PRINT " YOU MAY TAKE 1, 2, OR 3 STICKS"
2 MAKE
    NAME  "MOVE"
    THING  REQUEST
3 TEST MEMBER :MOVE "1 2 3"
4 IFFALSE OUTPUT GETMOVE
5 OUTPUT :MOVE
END
```

If the TEST is "FALSE", try again.

With these changes NIMPLAY is certainly a referee -- but still has some rough edges. For example, when :STICKS is 2, GETMOVE gives permission to take 1, 2, or 3 sticks! And if :PLAYER takes 3, :STICKS becomes negative and the game will go on forever on account of a SLIP-BY bug. However, we shall leave it as an exercise to remedy these minor failings.

In presenting this section to children we might work through one of the two major modifications with the class and let the children struggle with the other. The SLIP-BY bug we would leave to the class to discover and cure. Those who miss it at this stage will find its presence more obtrusive later -- and a profitable discussion might develop on the question of why the bug was not found -- perhaps, because the human player always makes reasonable moves so that :STICKS never becomes negative even though the machine would allow it. Later we shall see that when the machine makes its own moves it will not be so cooperative unless we tell it to be.

6.0. The Simplest Mechanical Player

How can the machine choose a move? The simplest way is by using RANDOM. For example, we could allow GETMOVE* the choice: if a person is to play use REQUEST, if the machine is to play use RANDOM. But it has to be told whether the player is human or the computer. So it must have an input.

```
TO GETMOVE :PLAYER
TEST IS :PLAYER  "COMPUTER"
IFTRUE  MAKE
          NAME   "MOVE"
          THING   RANDOM
IFFALSE  PRINT "YOU MAY TAKE 1, 2, OR 3 STICKS"
IFFALSE  MAKE
           NAME:  "MOVE"
           THING:  REQUEST
        .
        .
        .       (as before)
```

At this stage the SLIP-BY bug might become serious. One way to to kill it is to tell GETMOVE about :STICKS and have it try-again if :MOVE comes up greater than :STICKS . To do this we change the title line to:

```
TO GETMOVE :PLAYER  :STICKS
```

and add a pair of lines (in the TRY-AGAIN form) after the two MAKEs.

```
TEST GREATERP :MOVE  :STICKS
IFTRUE  OUTPUT GETMOVE :PLAYER  :STICKS
```

* Notice this anthropomorphism. We find it useful to talk of procedures as agents, of their "state of knowledge," of "telling them" of having them "talk to" one another. But we present this to children as a deliberate metaphor which they might find useful.

7.0. Strategic Play

Our plan for writing the NIM playing program in many strata now calls for it to recognize a few special numbers and know what to do in those cases, but continue to play stupidly in other cases. However, by this time it is likely that the class has already discovered the full strategy. It may still be worthwhile to encourage at least some members to follow the original plan as an instructive joke. In this section we shall illustrate a general question-answer technique for classroom discussion and to encourage habits of heuristic neatness in the children's own thinking.

7.1. A Semi-Smart NIM Player

A good exercise is to observe NIMPLAY in its present condition, and collect and classify its mistakes. An example of a classification made by a child is:

RETARD MISTAKES: There were 2 or 3 sticks and the machine did not take all!

DUMB MISTAKES: There were 5 sticks and the machine took 2. (If the machine had any sense it would leave the opponent with 4.)

If there are 6 or 7 it's dumb not to shoot for 4.

We shall write a procedure to avoid first "retard mistakes" and then "dumb mistakes".

Question: What program form?

Answer: TEST-TEST

Question: What do we test for?

English Answer: Whether there are 1, 2, or 3 sticks.

LOGO Answer: TEST MEMBER :STICKS "1 2 3"

We recall the procedure MEMBER shown by the examples:

MEMBER 6 "1 2 3" = "FALSE"

MEMBER 2 "1 2 3" = "TRUE"

Question: What is the action if the test is passed?

English Answer: Take all the sticks .

LOGO Answer: OUTPUT :STICKS

Question: What if it is not passed?

English Answer: Move just like before.

LOGO Answer: MAKE
NAME "MOVE"
THING RANDOM

Putting this together to make a procedure to make the move:

Question: What must the procedure know?

Answer: :STICKS -- so it needs an input.

Question: Operation or command?

Answer: Operation, because it will give us :MOVE as its output.

```
TO MAKEMOVE :STICKS

TEST MEMBER :STICKS "1 2 3"
IFTRUE OUTPUT :STICKS
IFFALSE OUTPUT RANDOM
END
```

MAKEMOVE is an easy name to remember.

The procedure is used in place of RANDOM in GETMOVE. So don't forget to change GETMOVE!

Now extra lines can be added. For example:

```
TEST IS :STICKS "5"

IFTRUE OUTPUT "1"
```

7.2. The Smart Player

By this time everyone should be very close to understanding the strategy, for example, in the following form:

Question: How does the game end?

Answer: When a player leaves zero sticks.

So let's try making the main actor be the number of sticks we leave. If we can leave zero that's great. But if we have more than 3 we can't. So we must think ahead.

Question: What can we leave so as to help us leave zero next time?

Answer: 4. Because the opponent will leave 1, 2, or 3.

Question: What can we leave so as to be able to leave 4 next time?

Answer: 8.

So 0, 4, 8 are good numbers to shoot at for leaving.

Question: What others?

Answer: 12, 16, ...

Question: Describe these.

Answer: REMAINDER :NUMBER: 4 = 0

REMAINDER :NUMBER :DIVIDER is an operation whose output is the remainder when :NUMBER is divided by :DIVIDER .

$64 Question: If I give you a number called :NUMBER , how can you use it to find the next number down divisible by 4?

Answer: Subtract REMAINDER :NUMBER 4.

So there we are! The smart invincible NIMplayer is made by replacing MAKEMOVE by SMARTMOVE.

```
TO SMARTMOVE :STICKS
MAKE
   NAME:  "REM"
   THING   REMAINDER :STICKS 4
TEST IS :REM  Ø
IFTRUE  OUTPUT 1

IFFALSE  OUTPUT :REM
END
```

This LOGOTHING is the main actor, so name it.

It really doesn't matter in this case.

8.0 Frills

Write superprocedures or make additions to the present procedures so that transcripts like the following will be produced:

```
NIM

DO YOU KNOW HOW TO PLAY NIM?
<NO
HERE ARE THE RULES:  YOU WILL BE SHOWN A CLOOLECTION OF X'S.  YOU MAY
REMOVE 1, 2 OR 3.  THE PLAYER WHO TAKES THE LAST WINS.  THIS IS
PROBABLY TOO VAGUE FOR YOU TO UNDERSTAND, BUT TRY PLAYING AND I'LL
CORRECT YOUR MISTAKES.

ARE YOU READY?
<I AM
PLEASE SAY "YES" OR "NO"
<YES
OK.  NOW TELL ME THE NAME OF THE FIRST PLAYER.
<JON
NOW THE NAME OF THE OTHER PLAYER
<COMPUTER
HOW MANY STICKS DO YOU WANT TO START WITH?
<THIRTY
I'M A DUMB COMPUTER.  TYPE A PROPER NUMERAL.
<31

JON TO PLAY.
THERE ARE 31 STICKS.
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
JON, TAKE 1, 2 OR 3
<3

COMPUTER TO PLAY.
THERE ARE 28 STICKS.
XXXXXXXXXXXXXXXXXXXXXXXXXXXX
I TAKE 3

JON TO PLAY.
THERE ARE 25 STICKS.
XXXXXXXXXXXXXXXXXXXXXXXXX
TAKE 1, 2 OR 3
<3
     .
     .
     .
```

8.1 Modifications

There are unlimited possibilities of "playing with" the ideas in the procedure after it has been made to work. The following three are merely examples to illustrate the idea that the project has not necessarily run out when the procedure is debugged.

An interesting simple modification to the rule of the game is to change the 1-2-3 rule to a 1-2 rule or a 1-2-3-4-5 rule. Write a procedure which will ask what rule is to be used.

Our stop rule was: the player who takes the last stick wins. Change this to: he who takes the last stick loses. (The latter is the traditional form; meeting a temporary change could be considered as part of planning for the project; students should be able to see and formulate the idea that our rule leads to a simpler algorithm without changing its principle.)

The game can be embedded in a more complex one, such as moving counters along marked paths on a board. If there is just one linear path, the problem is identical, but if branches are allowed, interesting complexities arise.

APPENDIX

A Listing of the NIMPLAY Procedures

```
TO NIMPLAY  :STICKS  :PLAYER  :OPPONENT
10  PRINT SENTENCE "THE NUMBER OF STICKS IS" AND  :STICKS
20  PRINT SENTENCE  :PLAYER AND
30  MAKE
      NAME  "NEWSTICKS"
      THING  :STICKS  -  GETMOVE  :PLAYER  :STICKS
40  TEST IS  :NEWSTICKS  Ø
50  IFTRUE PRINT SENTENCE  :PLAYER AND "IS THE WINNER"
60  IFTRUE STOP
70  NIMPLAY :NEWSTICKS  :OPPONENT  :PLAYER
END

TO GETMOVE  :PLAYER  :STICKS
10  TEST IS  :PLAYER  "COMPUTER"
20  IFTRUE MAKE
           NAME  "MOVE"
           THING  SMARTMOVE
30  IFFALSE PRINT  "YOU MAY TAKE 1, 2, OR 3 STICKS"
40  IFFALSE MAKE
            NAME  "MOVE"
            THING  REQUEST
50  TEST MEMBER :MOVE "1 2 3"
60  IFFALSE OUTPUT GETMOVE :PLAYER :STICKS
70  TEST GREATERP :MOVE :STICKS
80  IFTRUE OUTPUT GETMOVE :PLAYER :STICKS
90  OUTPUT :MOVE
END

TO SMARTMOVE
10  MAKE
      NAME  "REM"
      THING: REMAINDER :STICKS 4
TEST IS :REM Ø
20  IFTRUE OUTPUT 1
30  IFFALSE OUTPUT :REM
END
```

We include a listing of MEMBER, but assume that it was written before the NIM unit.

```
TO MEMBER :IT :LIST
10 TEST IS :LIST :EMPTY
20 IFTRUE OUTPUT "FALSE"
30 TEST IS :IT FIRST :LIST
40 IFTRUE OUTPUT "TRUE"
50 OUTPUT MEMBER :IT BUTFIRST :LIST
END
```