ABSTRACT

Student projects that involve writing generative grammars in the computer language, "LOGO," are described in this paper, which presents a grammar-running control structure that allows students to modify and improve the grammar interpreter itself while learning how a simple kind of computer parser works. Included are procedures for programing a computer to write postcards, sentences, poetry, and music: (1) draw a robot face, snowflakes, hydrocarbon structures, and hills: (2) introduce context sensitivity: (3) define number theory: and (4) parse or analyze word strings. (AEA)

ED204812

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

ARTIFICIAL INTELLIGENCE LABORATORY

AI MEMO 391

LOGO MEMO 39

GRAMMAR AS A PROGRAMMING LANGUAGE

Neil Rowe

October 1976

A B S T R A C T

This paper discusses some student projects involving generative grammars. While grammars are usually associated with linguistics, their usefulness goes far beyond just "language" to many different domains. Their application is general enough to make grammars a sort of programming language in their own right.

A simple grammar-running control structure is presented, uncomplicated and very suitable for student tinkering. So not only can students write grammars, but they can modify and improve the grammar interpreter itself, learning something about how a simple kind of computer parser works.

The work reported in this paper was supported in part by the National Science Foundation under grant number EC40708X and conducted at the Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, Mass.. The views and conclusions contained in this paper are those of the author and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the National Science Foundation or the United States Government.

CS206527

Contents

The first professor I saw was in a very large room, with forty pupils about him. After salutation, observing me to look earnestly upon a frame, which took up the greatest part of both the length and breadth of the room, he said perhaps I might wonder to see him employed in a project for improving speculative knowledge by practical and mechanical operations. But the world would soon be sensible of its usefulness, and he flattered himself that a more noble exalted thought never sprang in any other man's head. Every one knew how laborious the usual method is of attaining to arts and sciences; whereas by his contrivance the most ignorant person at a reasonable charge, and with a little bodily labour, may write books in philosophy, poetry, politics, law, mathematics, and theology, without the least assistance from genius or study. He then led me to the frame, about the sides whereof all his pupils stood in ranks. It was twenty foot square, placed in the middle of the room. The superficies was composed of several bits of wood, about the bigness of a die, but some larger than others. They were all linked together by slender wires. These bits of wood were covered on every square with paper pasted on them; and on these papers were written all the words of their language, in their several moods, tenses, and declensions, but without any order. The professor then desired me to observe, for he was going to set his engine at work. The pupils at his command took each of them hold of an iron handle, whereof there were forty fixed round the edges of the frame, and giving them a sudden turn, the whole disposition of the words was entirely changed. He then commanded six and thirty of the lads to read the several lines softly as they appeared upon the frame; and where they found three of four words together that might make part of a sentence, they dictated to the four remaining boys who were scribes. This work was repeated three or four times, and at every turn the engine was so contrived that the words shifted into new places, as the square bits of wood moved upside down.

Six hours a day the young students were employed in this labour; and the professor showed me several volumes in large folio already collected, of broken sentences, which he intended to piece together; and out of those rich materials to give the world a complete body of all arts and sciences; which however might be still improved, and much expedited, if the public would raise a fund for making and employing five hundred such frames in Lagado, and oblige the managers to contribute in common their several collections.

He assured me, that this invention had employed all his thoughts from his youth, that he had emptied the whole vocabulary into his frame, and made the strictest computation of the general proportion there is in books between the numbers of particles, nouns, and verbs, and other parts of speech.

Jonathan Swift, Travels into Several Remote Nations of the World (1726), III: 5

The painting machine had a wheel to hold a thousand smears of color and a brush mounted on a pivoted arm. The brush could be moved along the arm by one motor, while a second motor worked the arm around on its pivot. A third rotated the paint wheel, or "auto-palette."

Random numbers generated by the tape were fed into this system, controlling all three variables. The brush could dip up any color, transfer it to any of a hundred and fifty thousand positions over a prepared canvas, and dip again, leaving a dot. Between dottings, it moved through a powerful cleaning solution.

This cartesian process would go on until either the canvas was completely covered or until Ank liked what he saw and stopped it. He called the process rand-pointillisme in advance, knowing how important it was for his former colleagues to have a name to fasten upon from the start. Ank was prepared to explain in detail the philosophy behind this "marriage of random number and Seurat, which guarantees all the benefits of luminosity, color and harmony".

Now he set it into motion. There were a hundred and fifty million [sic] potential paintings in there somewhere, a hundred and fifty million pure abstract patterns without "meaning" or "intention". What he would see, in just a few hours, would be the end of so-called Humanism, the end of sentiment and prejudice -- the dawn of Mechanism.

What he actually saw was a close copy of David's Coronation of Napoleon. The details were blurry, but his painting differed from the original in only one respect.

The archbishop's face was modeled in bright greens.

Ank tried a fresh canvas. The brush rose and fell, faster than the eye could follow, and a "Remington" took shape: a mounted Indian wheeling his pony to fire an arrow into the flank of a galloping bison.

Except the pony wasn't wheeling and the bison did not gallop. Instead, both "stood on", or were solidified into, thick furry pedestals.

"Surrealism?" he whimpered. "I've given up my whole career for this cheap surrealism?"

It was almost time to go to Glen Dale's party. He threw the two ruined canvases in the corner and went to wash his hands. Instead of shaving, he decided to have a drink somewhere.

John Sladek, The Muller-Fokker Effect (1971), ch. 6

1. Introduction

This paper discusses some student projects involving generative grammars. While grammars are usually associated with linguistics, their usefulness goes far beyond just "language" to many different domains. Their application is general enough to make grammars a sort of programming language in their own right.

A simple grammar-running control structure is presented, uncomplicated and very suitable for student tinkering. So not only can students write grammars, but they can modify and improve the grammar interpreter itself, learning something about how a simple kind of computer parser works.

2. A one-command computer language

One way to explain the control structure is to think of it as a one-command computer language. Its one command is called R (for Rules). Since my experience has been with the language Logo, I will talk in terms of an implementation of this language within Logo.[1,2] (See Appendix for details of an implementation.)

The R ("rule" or "replace") command can be expressed as a function (or procedure) with two list arguments:

```
R [NAME] [JOE]
```

The command works always on a single list (string) of words. It replaces in the list particular words by other particular words. That above command, for instance, says to look for all occurrences of the word NAME in the list and change them to JOE. We can also replace one word by several:

```
R [NAME] [THE PRESIDENT OF THE UNITED STATES]
```

Oftentimes you're going to want to make choices when replacing stuff. That is, you won't always want to replace NAME with the same name JOE. We might want for variety to replace it occasionally with TOM, DICK, or HARRY. We can write this as follows:

```
R [NAME] [(JOE TOM DICK HARRY)]
```

The parentheses mean for the computer to choose only one of the things inside them. (To keep things simple, I assume random choice with each item having equal probability. If you want one item to be more likely than the others, put more than one instance of it into the list.)

We can combine the brackets and parentheses in commands:

```
R [NAME] [(JOE TOM DICK HARRY) C. (JONES DOE DOAKES)]
```

which replaces NAME by JOE C. JONES, HARRY C. DOE, and so on. Or we can put brackets within parenthesized expressions:

```
R [NAME] [(JOE [TERRIBLE TOM] [DICK THE INSURANCE SALESMAN] HARRY) C. (JONES DOE DOAKES)]
```

Remember, brackets mean use everything inside them; parentheses mean choose one and only one of the things inside them.

3. An example -- a postcard writing program

R commands by themselves aren't too interesting. You have to put several of them together. In Logo we can do this by defining a procedure. Here's a way to use the R language to write postcards, an idea suggested by a ninth grade student of mine:

```
TO POSTCARD
10 R [POSTCARD] [DEAR NAME , PHRASE . PHRASE . PHRASE . SIGNOFF , NAME]
20 R [NAME] [(TOM DICK HARRY SALLY SUE SANDRA OCCUPANT)]
30 R [PHRASE] [([HAVING A GREAT TIME] [WISH YOU WERE HERE] [WEATHER'S GREAT]
      [SURF'S UP] [BE SEEING YOU SOON] [CAN'T WAIT TO GET HOME])]
40 R [SIGNOFF] [(LOVE [BEST WISHES] [GOOD LUCK] [SO LONG FOR NOW])]
END
```

The control structure works on these R commands in the given order. It starts out with a list (string) consisting of one word, the name of the procedure (POSTCARD). It then goes down the list of R commands, making replacements of words in the string wherever it can. It then prints out the final list. Some of the "postcards" this procedure can produce include:

```
DEAR SALLY , BE SEEING YOU SOON . WISH YOU WERE HERE . WEATHER'S GREAT . GOOD LUCK , HARRY
DEAR TOM , CAN'T WAIT TO GET HOME . SURF'S UP . HAVING A GREAT TIME . BEST WISHES , DICK
DEAR OCCUPANT , WEATHER'S GREAT . HAVING A GREAT TIME . BE SEEING YOU SOON . LOVE, SANDRA
```

4. Writing sentences

We can also write more traditional "grammars". Here's a procedure to write some simple English sentences:

```
TO SIMPSENTENCE
10 R [SIMPSENTENCE] [NOUNPHRASE VERBPHRASE]
20 R [VERBPHRASE] [VERB NOUNPHRASE]
30 R [NOUNPHRASE] [([DETERMINER ADJECTIVE NOUN] NAME)]
100 R [VERB] [(LIKES HATES BOTHERS BEFRIENDS)]
110 R [DETERMINER] [(A THE SOME)]
120 R [ADJECTIVE] [(BIG TINY CHEERFUL SAD HAPPY GREEN PERPLEXED)]
130 R [NOUN] [(BOY GIRL COMPUTER ROBOT MARTIAN)]
140 R [NAME] [(TOM DICK HARRY SALLY SUE SANDRA)]
END
```

It can generate the following:

```
THE CHEERFUL ROBOT BOTHERS SOME SAD GIRL
TOM LIKES THE GREEN COMPUTER
A HAPPY MARTIAN BEFRIENDS SALLY
```

The nice thing is that you can add new features quite easily to this sentence generator. For instance, you can get sentences like

```
SOME BIG BOY IS SAD
A CHEERFUL ROBOT IS PERPLEXED
SANDRA IS CHEERFUL
```

by just changing line 20 to:

```
20 R [VERBPHRASE] [([VERB NOUNPHRASE] [IS ADJECTIVE])]
```

And, if you like, you can include adverbs in your sentences. Change line 20 to:

```
20 R [VERBPHRASE] [ADVERB ([VERB NOUNPHRASE] [IS ADJECTIVE])]
```

and add a line 90:

```
90 R [ADVERB] [(OFTEN SURPRISINGLY PERHAPS DUTIFULLY)]
```

Example:

```
DICK OFTEN IS HAPPY
A GREEN GIRL DUTIFULLY BEFRIENDS THE BIG ROBOT
```

Or suppose we want to allow an indefinite number of adjectives in front of the noun, like

```
THE BIG HAPPY GREEN ROBOT
```

which we can do by

```
30 R [NOUNPHRASE] [([DETERMINER ADJSTRING NOUN] NAME)]
35 R [ADJSTRING] [ADJECTIVE (ADJSTRING [])]
```

(The bracket pair [] means the "empty list" or the "list of no elements". If it is chosen instead of ADJSTRING, ADJSTRING will be replaced by only ADJECTIVE.)

Note that line 35 works because whenever a rule substitutes something in a sentence, it resumes searching just to the left of the inserted stuff. (That's to be sure to never "miss" a possible substitution.) So you can insert stuff into the inserted stuff and so on. Hence line 35 just keeps piling up adjectives in front of the noun until it manages to choose the second choice, the empty list.

Thus this allows us to get:

```
THE GREEN PERPLEXED ROBOT SURPRISINGLY LIKES SOME TINY SAD BOY
SUE PERHAPS HATES A BIG GREEN HAPPY MARTIAN
```

Finally, suppose we want to have compound sentences, sentences composed of two subsentences joined by a word like "and". Change line 10 to read

```
10 R [SIMPSENTENCE] [NOUNPHRASE VERBPHRASE ([] [(AND BUT SINCE THOUGH)
SIMPSENTENCE])]
```

giving:

```
THE TINY HAPPY ROBOT OFTEN LIKES SANDRA AND TOM SURPRISINGLY IS SAD
```

and so on. There are many possibilities for further development.

So in summary here's our new improved sentence generator:

```
TO SIMPSENTENCE
10 R [SIMPSENTENCE] [NOUNPHRASE VERBPHRASE ([] [(AND BUT SINCE THOUGH)
SIMPSENTENCE])]
20 R [VERBPHRASE] [ADVERB ([VERB NOUNPHRASE] [IS ADJECTIVE])]
30 R [NOUNPHRASE] [([DETERMINER ADJLIST NOUN] NAME)]
35 R [ADJLIST] [ADJECTIVE ([] ADJLIST)]
40 R [VERB] [(LIKES HATES BOTHERS BEFRIENDS)]
50 R [DETERMINER] [(THE A SOME)]
60 R [ADJECTIVE] [(BIG TINY HAPPY SAD GREEN PERPLEXED)]
70 R [NOUN] [(BOY GIRL COMPUTER ROBOT MARTIAN)]
80 R [NAME] [(TOM DICK HARRY SALLY SUE SANDRA)]
90 R [ADVERB] [(OFTEN SURPRISINGLY PERHAPS DUTIFULLY)]
END
```

5. Writing poetry

Consider the problem of writing poems in which the last syllables of the line must rhyme. We could try:

```
TO LIMERICK
10 R [LIMERICK] [A A B B A]
20 R [A] [DOWN UP DOWN UP DOWN RHYME1]
30 R [B] [DOWN UP DOWN RHYME2]
40 R [RHYME1] [(DATE FATE WAIT SATE SMELL BELL HELL WELL BROKE COKE JOKE YOKE)]
50 R [RHYME2] [(FEAT BEAT SEAT HEAT WAY SAY BAY HAY MOOD FOOD STEWED RUDE)]
60 R [DOWN] [(UH AH ER IR AN UN IS US AW E)]
70 R [UP] [(MEAN PROTE VAST SPRILL TRAMMED SLOOSED POUNT GRASP DRUNK)]
END
```

but this runs into a problem: each time the interpreter substitutes for RHYME1 or RHYME2, it will choose a new word. So we have no way of ensuring that all the A lines or all the B lines will have the <u>same</u> rhyme. That is, we have no way to force a rhyme.

It seems what we need is a "new R command", call it R.ONCE, that works just like the old, except it only chooses once. (See Modification #1 in the Appendix.) Using it we can rewrite our limerick-writing program this way:

```
TO LIMERICK
10 R [LIMERICK] [A A B B A]
20 R [A] [DOWN UP DOWN UP DOWN RHYME1]
30 R [B] [DOWN UP DOWN RHYME2]
40 R.ONCE [RHYME1] [(ATE ELL OKE)]
50 R.ONCE [RHYME2] [(EAT AY OOD)]
100 R [ATE] [(DATE FATE WAIT SATE)]
110 R [ELL] [(SMELL BELL HELL WELL)]
120 R [OKE] [(BROKE COKE JOKE YOKE)]
130 R [EAT] [(FEAT BEAT SEAT HEAT)]
140 R [AY] [(WAY SAY BAY HAY)]
150 R [OOD] [(MOOD FOOD STEWED RUDE)]
200 R [DOWN] [(UH AH ER IR AN UN IS US AW E)]
210 R [UP] [(MEAN PROTE VAST PRILL TRAMMED SLOOSED POONT GRASP DRUNK)]
END
```

A sample limerick:

```
AH GRASP UN POONT E DATE
AN SLOOSED IR POONT UH SATE
ER VAST AN FOOD
UN PRILL IS STEWED
AW PROTE IR TRAMMED US FATE
```

Our Logo system has a speech synthesizer, so we can generate actual sounds (as in the case of the above) by figuring out the phonemes necessary for each word.

6. Writing music

It's easy to extend these ideas to music. Let's consider a situation in which we're only concerned with specifying the pitch and durations of musical notes. We can specify the pitch as a letter -- assuming the range of an octave, that means letters C, D, E, F, G, A, B, and an upper C which we can call CC. The duration can be either a Q (quarter), H (half), DH (dotted half), or W (whole) note.

Then we can represent a melody by a string. For instance,

[C H E Q A Q]

represents a C half note followed by E and A quarter notes.

So here's a program that writes melodies according to a few simple harmonic schemes. It first chooses a harmony for each measure, then constructs measures to fit that harmony. To make its melody a little more unified, it uses R.ONCE to make sure that measures with the same harmony have the same rhythm.

```
    TO MELODY
    10 R [MELODY] [CCHORD (CCHORD GCHORD FCHORD) CCHORD ([CCHORD GCHORD DCHORD]
[GCHORD DCHORD GCHORD] [GCHORD CCHORD FCHORD]) GCHORD CCHORD].
    20 R.ONCE [CCHORD] [([CNOTE W] [CNOTE DH CNOTE Q] [CNOTE H CNOTE Q CNOTE Q])]
    30 R.ONCE [GCHORD] [([GNOTE H GNOTE H] [GNOTE Q ORDNOTE Q GNOTE Q ORDNOTE Q])]
    40 R.ONCE [FCHORD] [([FNOTE H FNOTE H] [FNOTE Q ORDNOTE Q FNOTE Q ORDNOTE Q])]
    50 R.ONCE [DCHORD] [([DNOTE H DNOTE H] [DNOTE Q ORDNOTE Q DNOTE Q ORDNOTE Q])]
    100 R [CNOTE] [(C E G CC)]
    110 R [GNOTE] [(D G B)]
    120 R [FNOTE] [(C F A CC)]
    130 R [DNOTE] [(D F A)]
    140 R [ORDNOTE] [(D E F G A B)]
    END
```

Sample melodies are given in Fig. 1. With our system you can take such a melody and play sounds for it via a "music box".

7. Drawing a robot face

We can apply the idea of a grammar to drawing designs too. Consider something called a "turtle" that lives on a surface of something like a television picture tube. Suppose he can do two things: he can move forward a specified distance, leaving a line behind him, or he can turn right a specified number (either positive or negative) of degrees. These operations I will call "FORWARD" and "RIGHT"(which can be abbreviated "FD" and "RT"). (He can only do one of those at a time.)

So, for example, these are the commands you would give the turtle to draw a square:

FD 10 RT 90 FD 10 RT 90 FD 10 RT 90 FD 10 RT 90

But it's hard having to draw pictures with your pen constantly down. For this reason, the turtle also has a command called "PENUP". It allows him to move around just the same as always, except that he won't leave any line behind him. Normal mode is restored by a command called "PENDOWN".

So let's write a grammar to draw "robot" faces. We'll use a large square for the head, small squares for the eyes, and small rectangles in a row for the teeth. We make choices as to whether to have a tall head or a square head, have the eyes and mouth high or low in the head, and whether to show the teeth in the mouth.

```
TO FACE
10 R [FACE] [HEAD (SETHIGH SETLOW) EYES MOUTH]
20 R [HEAD] [(TALLHEAD SQUAREHEAD)]
100 R [EYES] [EYE SETUPEYE2 EYE]
200 R [MOUTH] [SETUPMOUTH (OPENMOUTH SIXTEETH)]
210 R [SIXTEETH] [TOOTHPLUS TOOTHPLUS TOOTHPLUS TOOTHPLUS TOOTHPLUS TOOTHPLUS]
220 R [TOOTHPLUS] [TOOTH PENUP FD 10 PENDOWN]
300 R [SETHIGH] [PENUP FD 80 RT 90 FD 20 PENDOWN]
310 R [SETLOW] [PENUP FD 50 RT 90 FD 20 PENDOWN]
320 R [SETUPEYE2] [PENUP FD 40 PENDOWN]
330 R [SETUPMOUTH] [PENUP FD 19 RT 90 FD 55 RT 90 PENDOWN]
END
```

where the remaining undefined words are just rectangles and squares of various sizes. They can be defined by a length and a width:

TALLHEAD as 140 by 100
SQUAREHEAD as 100 by 100
EYE as 20 by 20
OPENMOUTH as 60 by 15
TOOTH as 10 by 15

Sample faces are given in Fig. 2.

8. Drawing snowflakes

Now let's write a grammar to draw so-called "snowflake curves", by making up a list of FD and RT instructions, and then executing them in sequence. "Snowflake curves" are a progressive sequence of drawings like those in Fig. 3. They follow rules something like this:

```
TO FLAKE
10 R [FLAKE] [SIDE R SIDE R SIDE R]
20 R.ONCE [SIDE] [([FD 1] [SIDE L SIDE R SIDE L SIDE])]
END
```

where R stands for "RT 120" and L stands for "RT -60" (which is the same as turning left 60).

Line 20 says that at any point in the process, either make all the SIDEs straight lines or else elaborate all of them. But this runs into a curious problem: if we take the second choice in that line, we'll never get out of line 20, because with every substitution for SIDE four new SIDEs are added that must also be substituted for! (Like Hercules and the Hydra.) But on the other hand, if we made line 20 a R rather than R.ONCE command, we would be getting asymmetrical snowflakes, which isn't what we want either.

We could avoid this problem if we just never again touched things we substituted into the grammar. (See Appendix, Modification #2.) But this runs into a further problem that we'll never come back to line 20 after we're through with it, and line 20 might have left SIDEs in the string. So modify the grammar control structure so that lines of the grammar get "second chances". We'll still keep the idea of applying the rules in the specified order, but when we come to the end of them, we'll go back to the beginning again. So we'll repeatedly "cycle" through until there's nothing left to substitute for. (See Appendix, Modification #3).

So now the FLAKE we originally wrote works.

9. Drawing hydrocarbons

We can use grammars to explore some aspects of chemical structure. Consider the following grammar for drawing some hydrocarbons:

```
TO HYDROCARBON
10 R [HYDROCARBON] [MARK "C" CHAIN LT 90 CHAIN LT 90 CHAIN LT 90 CHAIN LT 90]
20 R [CHAIN] [(HATOM HATOM HATOM HATOM HATOM CATOM C2ATOM)]
30 R [HATOM] [SHORTDASH MARK "H" RT 180 SHORTDASH]
40 R [CATOM] [DASH MARK "C" LT 90 CHAIN LT 90 CHAIN LT 90 CHAIN LT 90 DASH]
50 R [C2ATOM] [DASH MARK "C" LT 90 CHAIN LT 90 DOUBLEDASH MARK "C" LT 90 CHAIN LT 90
CHAIN DOUBLEDASH DASH]
END
```

where MARK is a command that draws a letter, and where DASH, SHORTDASH, and DOUBLEDASH are defined in the obvious way.

The grammar builds up a molecule by starting with a carbon atom and attaching to each of its four sides either a hydrogen atom, a single-bonded carbon atom, or a double-bonded carbon atom pair. In the case of the last two, the process is repeated for bonds of those carbon atoms.

Note that since the entire molecule must be drawn by a step-by-step process, "backing up" must be done at times -- when you draw an H, you must back up to the center of the attached C. That's the reason for the extra SHORTDASH, DASH, and DOUBLEDASH in lines 30, 40, and 50 -- they're just ways of backing up. The easy way to do this in this case is just to redraw the dashes going the other direction, since they're all symmetrical.

Some sample chemical structures are given in Fig. 4.

10. Drawing hills

Let's write a grammar for drawing different sizes of "hills" -- that is, lines that slope up, then down in a symmetrical way. We could try:

```
TO HILL
10 R [HILL] [RT -45 PEAK RT 45]
20 R [PEAK] [FD 10 (PEAK [RT 90]) FD 10]
END
```

which gives "hills" like those in Fig. 5.

But now what about making the slopes more gradual, like real hills? One way might be to follow a set of commands like this:

```
U U U D D D D D D U U U
```

where "U" stands for the upward-curving arc "FD 2 RT -5" and "D" stands for the downward-curving arc "FD 2 RT 5". Fig. 6 shows a few of this type of hill.

For a grammar, this suggests (assuming the original control structure, without the modifications):

```
TO HILL
10 R [HILL] [U D (HILL []) D U]
END
```

(Remember, the [] represents a list of no words at all. So if the random choice chooses it, HILL will be replaced by [U D D U].)

But this doesn't work. It generates the U's and D's alternately, like

```
U D U D U D D U D U D U
```

instead of what we want:

```
U U U D D D D D D U U U
```

So we could try (assuming Modification #3, cyclic rule application):

```
TO HILL
10 R [HILL] [UPSLOPE DOWNSLOPE]
20 R [UPSLOPE] [U (UPSLOPE []) D]
30 R [DOWNSLOPE] [D (DOWNSLOPE []) U]
END
```

but that means that the two slopes can be of different heights, as for instance

```
U U U U U U U U D D D D D D D D D D U U
```

which isn't what we want either. Is there a way out?

11. Introducing context-sensitivity

It seems we've come up against a basic limitation of our grammar interpreter. That is, we can by the R.ONCE force the grammar to make a consistent choice (expansion) in several instances of the same symbol. This is one way of getting coordinated substitutions. But we cannot make one choice affect another. That's what we need in this hill example -- we have to create two independent types of symmetry.

What we need is some way to restrict the application of rules to only particular contexts. The simplest way might be to make the first argument to the R and R.ONCE commands, which represents the stuff we're looking to match, be <u>more than one word</u>. (See Appendix, Modification #4.) That way we could write

```
R [U MIDSLOPE] [U U MIDSLOPE D]
```

meaning that we want every occurrence of MIDSLOPE that is preceded by an U to have another U inserted in front of it, and another D inserted after it. So we could write the hill-drawing program this way (assuming that Modification #1 (R.ONCE), Modification #2 (no immediate replacement of substituted words), and Modification #3 (cyclic rule application) are still in effect):

```
TO HILL
10 R [HILL] [U MIDSLOPE D D MIDSLOPE U]
20 R.ONCE [MIDSLOPE] [([] MIDSLOPE)]
30 R [U MIDSLOPE] [U U MIDSLOPE D]
40 R [D MIDSLOPE] [D D MIDSLOPE U]
END
```

where, as before, "U" stands for "FD 2 RT -5" and "D" for "FD 2 RT 5". We can indeed now draw the hills of Fig. 6.

12. Number theory

As a final example of what we can do with grammars, note that some number-theoretic ideas can be defined by them. For example, we can generate all odd powers of two by a one-line grammar:

```
TO ODDPOWER2
10 R.ONCE [ODDPOWER2] [([2] [2 * 2 * ODDPOWER2])]
END
```

Or write out strings consisting of an odd power of 2 number of X's:

```
TO ODDP2
10 R.ONCE [ODDP2] [([X X] [ODDP2 ODDP2 ODDP2 ODDP2])]
END
```

Or you could generate members of the Fibonacci series, using our HILL grammar as a model (assuming all Modifications in effect except #2):

```
TO FIBO
10 R [FIBO] [A + B]
20 R.ONCE [A] [(1 NEWB)]
30 R [1 + B] [1 + 1]
40 R [B + 1] [1 + 1]
50 R [B] [FIBO]
60 R [NEWB] [B]
END
```

The final string produced will be alternating 1's and +'s, like 1 + 1 + 1 + 1 + 1. While it is being generated, the string consists of A's and B's alternating with +'s. The number of A's represents the nth Fibonacci number, the number of B's the (n+1)th Fibonacci number, and hence the total number of letters the (n+2)th Fibonacci number.

Line 20, the only line where a choice is made, decides whether to replace all symbols by 1's now or go on to generate the next largest Fibonacci number. Lines 30 and 40 are just to ensure that when you are finishing the generation and 1 is the selection in line 20 (i.e., all A's are changed into 1's), all B's will be changed into 1's too.

13. Additional projects

Try writing grammars for the following.

1. Simple stories, e.g. fables
2. Stereotyped newspaper stories
3. Advertisements
4. Mantras
5. Something like SIMPSENTENCE but with subject-verb agreement in number (singular vs. plural)
6. Or ability to use "an" and "a" properly
7. Or ability to substitute a pronoun (the correct one) occasionally
8. Simple sentences in some foreign language
9. Simple dialogues between two or more people (e.g. plays).

10. Musical melodies based on melodic (as opposed to harmonic) considerations. For instance, consider which notes of the scale sound good after a particular other note of the scale..

11. Passacaglias on a given ground; that is, music with a bass (lowest) part that consists of a short melody repeated over and over

12. Rondos where the sections are all in ternary form; that is, music consisting of a single section alternating with other sections (as for instance ABACADA), where each section consists of three parts, the first and last parts being identical (ABA)

13. Different kinds of simple houses

14. Apartment houses of random size and shape, with shades drawn for random windows, plants in the window for random windows, etc.

15. Space-filling curves; that is, given a square region of fixed size, a line within it such that any point within the square is closer than some small fixed distance away from the line.

16. Trees and bushes. Find out something about the way real ones grow (like how far between branches, or what angles the branches are likely to grow out at), and try to model it.

17. A different kind of chemical structure. Try using context-sensitive rules to eliminate chemically impossible or unlikely configurations.

18. Simple particle physics. That is, try to create bubble chamber particle tracks. For example, a neutron (moving in a straight line) hits another particle and breaks up into a proton (moving in a clockwise curve) and an electron (moving in a counterclockwise curve), both of which eventually decay into something else. (Note that some particles are invisible.)

19. Some kind of electronic circuit diagrams, perhaps digital logic

20. The rows of Pascal's triangle

21. Composite (not prime) numbers

22. "Agendas" for your daily activities

14. Further control structure modifications

As you may observe, one of the nicest things about this system is the relative ease of making changes to the control structure (or interpreter), thanks to its relative simplicity. This paper has introduced four significant improvements to the original "bare bones" interpreter: the R.ONCE feature, preventing rules from reworking just-substituted words, cyclic rule application, and simple context-sensitivity. Many further projects are suggested, some from current work in linguistics.[3,4]

For one, it might be nice to have a "wild-card" symbol that will match anything. That is, assuming * to be that symbol, we could rewrite HILL to be this way:

```
TO HILL
10 R [HILL] [L MIDUP R R MIDDOWN L]
20 R [MIDUP * MIDDOWN] [([*] [L MIDUP R * R MIDDOWN L])]
END
```

where * will match whatever is between the MIDUP and MIDDOWN.

Extending this, we might like to specify for part of the matched pattern, not just anything, and not just a single word, but something in between. Like for a sentence generating program:

```
R [*ADJECTIVE] [(INCREDIBLY AMAZINGLY FRIGHTENINGLY) *ADJECTIVE]
```

where *ADJECTIVE matches anything that is an adjective.

A very powerful idea that might be used is that of the linguistic transformation. This means rules that work on strings but take into account how the strings were generated (their "derivational"). An example would be the "passive transformation", as in the following crude form:

```
R [*NOUNPHRASE1 *VERB *NOUNPHRASE2] [*NOUNPHRASE2 BE *VERB BY *NOUNPHRASE1]
```

which takes whatever the NOUNPHRASE1 has been expanded to and makes it the object of an agent prepositional phrase, and takes whatever NOUNPHRASE2 has been expanded to and makes it the subject. So for instance

THE BIG PINK ROBOT HATES NASTY BOYS

could become, after applying tense rules to change BE to ARE:

NASTY BOYS ARE HATED BY THE BIG PINK ROBOT

Adding this facility involves some challenging problems.

15. Parsing: turning the grammar around

An interesting project is to "turn the grammar around" and use it to analyze strings of words, rather than create them. For instance, is a given simple sentence grammatical English? Or does a given picture of a face show teeth?

A way to do this is just "run a grammar backwards". That is, you start with a string and the last rule of a grammar. You then try to find a match between things in your string and the <u>second</u> argument to the R command. (If the second argument contains choices, try every possibility.) If you find a match, substitute the <u>first</u> argument. This approach works fine for many context-free grammars.

16. Educational utility

I have tried here to give a concrete example of what has been called "learner-controlled computing".[5] How useful is it educationally? One incident may be revealing. When I first introduced this system to the author of the postcard-writing program given at the beginning of this paper, I had him write a grammar for simple sentences. He was studying English and German at his school, so he had a fair exposure to what is referred to in the schools as "grammar". So I said, "We need some kind of sentence pattern. How about noun-verb-noun?"

It sounded familiar to him. So he wrote

```
TO S
10 R [S] [NOUN VERB NOUN]
20 R [NOUN] [(PEN BOOK CAT DOG)]
```

and then said, "What's a verb?"

"An action word," I said, assuming that a hint would be sufficient. He looked a little mystified, but I prompted to go ahead and try something out, since he could easily change it later. So he wrote

```
30 R [VERB] [(IS DULL HARD HOT ANGRY)]
```

ran his program, and got back

```
BOOK HARD CAT
DOG ANGRY PEN
CAT HOT BOOK
. . . . . . . . . . . . .
```

Sentences generated by <u>his own rules</u> stared up at him from the page. He was surprised, and a little amused. And he began to think about what a verb really was, something which, despite his survival of many years of formal education, he hadn't really come to grips with. Verbs, as in fact nouns, must be "action words", concepts defined by a relationship within a sentence.

This is a lot healthier approach to grammar than any amount of ultimately arbitrary definitions. And I think it leads to a better understanding of what a verb really is.

Appendix: a Logo implementation

The overall structure of the components (procedures) is like this:

$$\begin{tikzcd}
\text{SAY} \arrow[d, dash] \\
\text{user's procedure (grammar)} \arrow[d, dash] \\
\text{R} \arrow[d, dash] \\
\text{MATCHREPLACE} \arrow[d, dash] \\
\text{EXPANSION} \arrow[d, dash] \\
\text{RANDCHOICE} \arrow[d, dash] \\
\text{ITEM}
\end{tikzcd}$$

SAY is the top-level procedure. It sets the initial word list to the list consisting of one name, the name it is called with. It then applies the rules in order to this list, making the necessary substitutions.

```
TO SAY :PROCNAME
10 MAKE "STRING :PROCNAME
20 RUN :PROCNAME
30 PRINT :STRING
40 SAY :PROCNAME
END
```

R is the procedure that executes a particular grammar rule. It replaces in the :STRING list all occurences of :SYMBOL by :REPLACEMENT.

```
TO R :SYMBOL :REPLACEMENT
10 MAKE "STRING MATCHREPLACE :STRING :SYMBOL
END
```

MATCHREPLACE is the workhorse of the grammar. It goes through the :STRING list word by word. If it finds an exact match between the :SYMBOL and a word of :STRING, it replaces that word by the :REPLACEMENT list. (Note that :REPLACEMENT is a free variable, not an argument, in this procedure, to save a little argument-passing.) It then goes to the beginning of the substitution and resumes the search process from there.

```
TO MATCHREPLACE :STRING :SYMBOL
10 IF (EMPTYP :STRING) OUTPUT :STRING
20 TEST ((F :SYMBOL) = (F :STRING))
30 IFTRUE OUTPUT (MATCHREPLACE SENTENCE (EXPANSION :REPLACEMENT) (BUTFIRST :STRING)
:SYMBOL)
40 IFFALSE OUTPUT SENTENCE (FIRST :STRING) (MATCHREPLACE (BUTFIRST :STRING) :SYMBOL)
END
```

EXPANSION expands the "replacement" (second) part of the R rule. It forms a single simple list for substitution into :STRING. For parenthesized expressions it makes a random choice as to which item to use; for brackets it takes the whole list within the brackets. (Note that the procedure assumes that a sublist just inside a bracketed list is parenthesized, and vice versa; it doesn't actually check.)

```
        TO EXPANSION :STRING
        10 IF EMPTYP :STRING THEN OUTPUT []
        20 TEST LISTP FIRST :STRING
        30 IFFALSE OUTPUT SENTENCE (FIRST :STRING) (EXPANSION BUTFIRST :STRING)
        40 IFTRUE OUTPUT SENTENCE (EXPANSION RANDCHOICE FIRST :STRING) (EXPANSION BUTFIRST
:STRING)
        END
```

RANDCHOICE figures out the length of a list, generates a random number from 1 up to that number, and outputs that numbered item of the list. (RANDOM :X :Y outputs a random integer of the range :X through :Y.)

```
        TO RANDCHOICE :LIST
        10 OUTPUT ITEM (RANDOM 1 (COUNT :LIST)) :LIST
        END
```

ITEM outputs the Nth item of list L.

```
        TO ITEM :N :L
        10 IF EMPTYP :L THEN OUTPUT []
        20 IF (:N < 2) THEN OUTPUT FIRST :L
        30 OUTPUT ITEM (:N - 1) (BUTFIRST :L)
        END
```

Modification #1: substituting the same choice in all places

Write a new procedure:

```
        TO R.ONCE :SYMBOL :REPLACEMENT
        10 MAKE "REPLACEMENT (EXPANSION :REPLACEMENT)
        20 MAKE "STRING MATCHREPLACE :STRING :SYMBOL
        END
```

This works because if we expand the :REPLACEMENT list before calling MATCHREPLACE (the string searching procedure), the resulting list will consist of no sublists, and hence cannot be expanded further.

Modification #2: avoiding changing substituted stuff

It is just necessary to change the line in MATCHREPLACE, line 30, which determines what to work on next after a match is found. Change it to:

```
        30 IFTRUE OUTPUT SENTENCE (EXPANSION :REPLACEMENT) (MATCHREPLACE (BUTFIRST :STRING)
:SYMBOLS)
```

assuming Modification #1 to have also been made.

Modification #3: cyclic rule application

To do this, write these two new outer procedures:

```
TO DOIT :PROCNAME
10 MAKE "STRING :PROCNAME
20 CYCLETHRU :PROCNAME
30 RUN :STRING
40 DOIT :PROCNAME
END

TO CYCLETHRU :PROCNAME
10 MAKE "CHANGEFLAG "FALSE
20 RUN :PROCNAME
30 IF (:CHANGEFLAG = "TRUE) CYCLETHRU :PROCNAME
END
```

And change MATCHREPLACE so as to set the flag to "TRUE whenever a substitution is actually made in the string:

```
25 IFTRUE MAKE "CHANGEFLAG "TRUE
```

So CYCLETHRU will stop wherever not a single rule of the grammar was applied on execution of the grammar procedure.

<u>Modification #4: matching for more than one word</u>

We can just modify MATCHREPLACE to handle a :SYMBOL which is more than one word (here assuming Modification #2 and #3 still in effect). COUNT gives the number of items in a list.

```
TO MATCHREPLACE :STRING :SYMBOLS
10 IF ((COUNT :STRING) < (COUNT :SYMBOLS)) OUTPUT :STRING
20 TEST MATCHP :STRING :SYMBOLS
25 IFTRUE MAKE "CHANGEFLAG "TRUE
30 IFTRUE OUTPUT SENTENCE (EXPANSION :REPLACEMENT) (MATCHREPLACE (DROPNUM (COUNT
:SYMBOLS) :STRING) :SYMBOLS)
40 IFFALSE OUTPUT SENTENCE (FIRST :STRING) (MATCHREPLACE (BUTFIRST :STRING) :SYMBOLS)
END
```

Procedure MATCHP checks to see if :SHORTSTRING exactly corresponds to the beginning of :BIGSTRING.

```
TO MATCHP :BIGSTRING :SHORTSTRING
10 IF (EMPTYP :SHORTSTRING) THEN MAKE "CHANGEFLAG "TRUE OUTPUT "TRUE
20 TEST ((FIRST :BIGSTRING) = (FIRST :SHORTSTRING))
30 IFTRUE OUTPUT MATCHP (BUTFIRST :BIGSTRING) (BUTFIRST :SHORTSTRING)
40 IFFALSE OUTPUT "FALSE
END
```

Procedure DROPNUM outputs :LST with the specified number of items dropped off its front end.

```
TO DROPNUM :NUMITEMS :LST
10 IF (:NUMITEMS < 1) THEN OUTPUT :LST
20 OUTPUT DROPNUM (:NUMITEMS - 1) (BUTFIRST :LST)
END
```

Acknowledgements

The ideas presented here are not particularly original. There is a large body of knowledge regarding grammars and parsing within computer science. The idea of writing grammars in Logo as a student programming project is due to Ken Kahn.[6] He constructed a system similar to, but more limited than that described here, to provide a framework for generating English sentences. I have tried to extend and clarify his work, in particular by rewriting the interpreter to make it more accessible to student understanding and tinkering.

Another major influence has been the work of Ira Goldstein and Mark Miller[7] in specifying a grammar for a broad class of programming processes. This work emphasizes the grammatical nature of programming. Mention should also be made of the "production system" model of Allen Newell and Herbert A. Simon.[8]

As far as specific precedents, there is SNOBOL, a computer language containing as a subset several facilities for grammar-like activities.[9] However, SNOBOL is not primarily an interactive language. Its design bias emphasizes code efficiency, not language usability. These features tend to make it unsuitable for educational use.

There is also specific work detailing methods of employing grammars in particular domains.[10,11,12,13,14] And finally, I must acknowledge the work of Seymour Papert and others in developing a new kind of learning environment based around the use of the computer language Logo.

Thanks to Hal Abelson, Andy DiSessa, Ken Kahn, and Mark Miller for help with this paper.

References

1. Seymour Papert, "Uses of Technology to Enhance Education", MIT Artificial Intelligence Laboratory Logo Group Memo #8, June 1973.

2. Hal Abelson, Nat Goodman, and Lee Rudolph, "Logo Manual", MIT Logo Memo #7, June 1974; or contact the author for a draft of his manual in preparation.

3. Emmon Bach, Syntactic Theory, Holt, Rinehart, & Winston, 1973.

4. Andreas Koutsoudas, Writing Transformational Grammars: An Introduction, McGraw-Hill, 1966.

5. Stuart D. Milner, "Learner-Controlled Computing: A Description and Rationale", Journal of Educational Technology Systems, Winter 1974, p.207.

6. Ken Kahn, "A Logo Natural Language System", MIT Logo Group Working Paper #46, December 3, 1975.

7. Ira Goldstein and Mark Miller, "Intelligent Tutoring Programs: A Proposal For Research", MIT Logo Group Working Paper #50, 1976.

8. Allen Newell and Herbert A. Simon, Human Problem Solving, Prentice-Hall, 1972.

9. James F. Gimpel, Algorithms in SNOBOL4, Wiley, 1976.

10. William A. Woods, "Mathematical Linguistics and Automatic Translation", Harvard University Aiken Computation Laboratory, Report No. NSF-19 (September 1967).

11. David E. Rumelhart, "Notes on a Schema for Stories", in Representation and Understanding, Bobrow and Collins, ed., 1975, p.211.

12. Gahan Wilson, "The Science Fiction Horror Movie Pocket Computer", in National Lampoon: The Paperback Conspiracy, Warner, 1974.

13. Terry Winograd, "Linguistics and Computer Analysis of Tonal Harmony", Journal of Music Theory, 12:1 (1968), p.2.

14. A. C. Shaw, "A Formal Picture Description Scheme as a Basis For Picture Processing Systems", Information and Control, 14, 1969, p.5

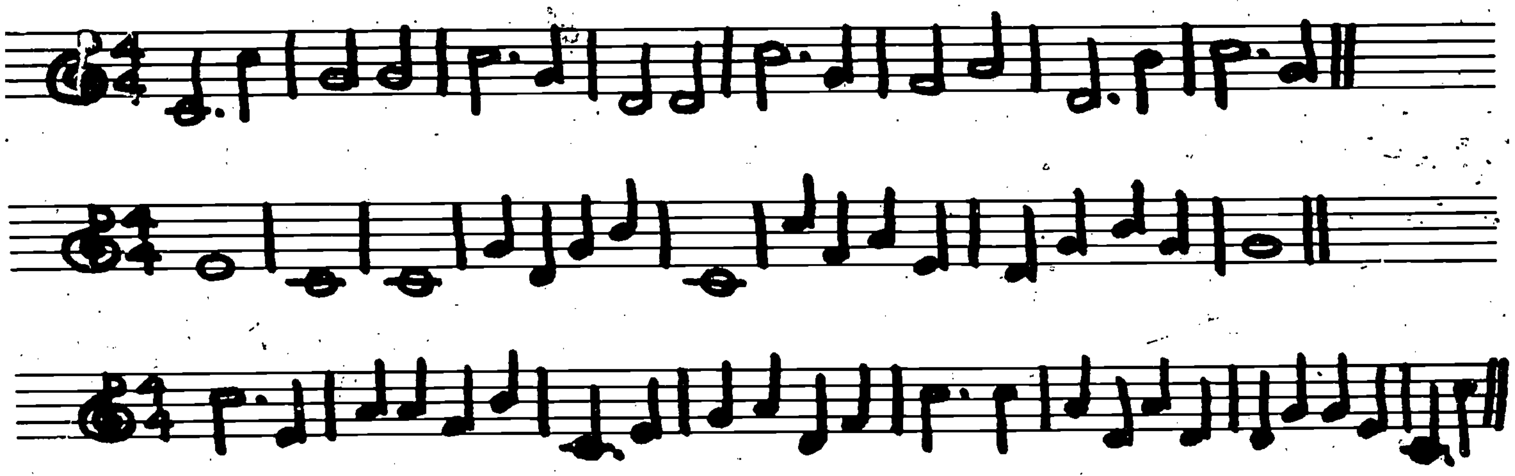

Fig. 1

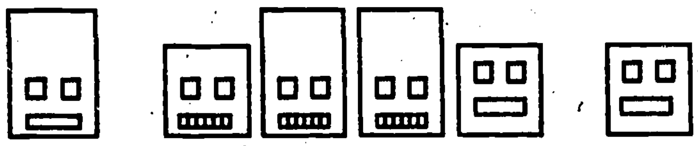

Fig. 2

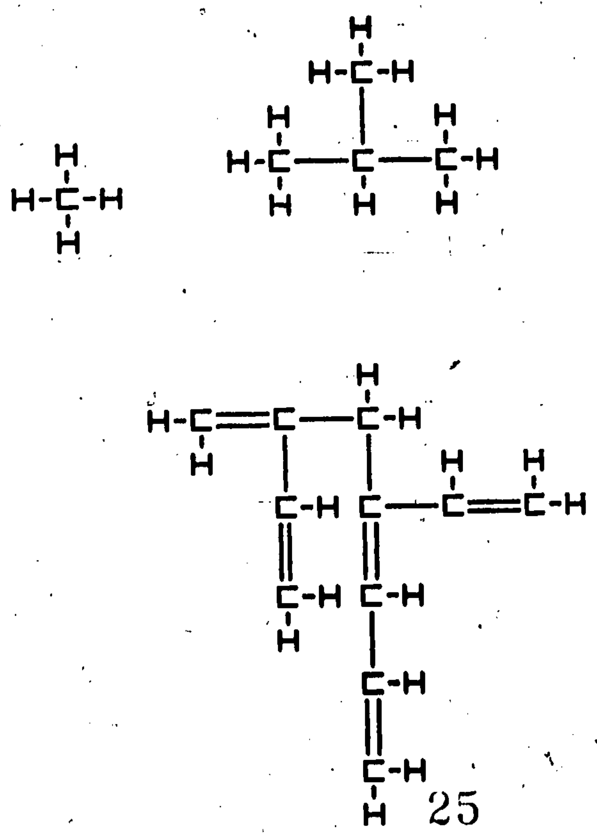

Fig. 3

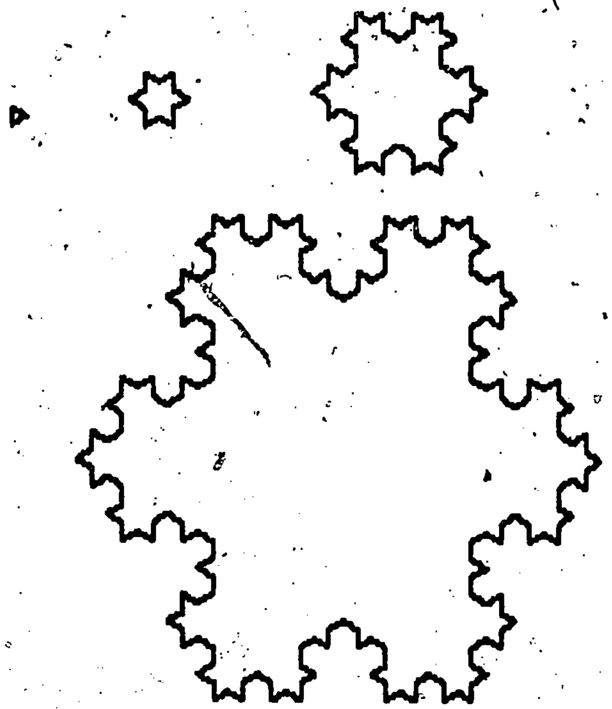

Fig. 4

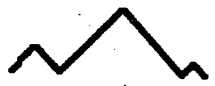

Fig. 5

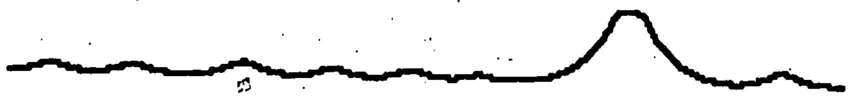

Fig. 6