GRIP—Michael's sections

Mark C. Torrance
Michael P. Frank
Department of Computer Science
Massachusetts Institute of Technology
Cambridge, Massachusetts 02139

June 17, 1992

Abstract

We describe GRIP.

1 Introduction

GRIP stands for Grid Processing, in analogy with LISP. GRIP is a graphical programming language based on 2-dimensional Turing machines, as discussed in [1]. I like GRIP.

Imagine a large grid of square cells. In analogy with the standard Turing machine, one can imagine a 2-dimensional form of Turing machine, which was actually the kind that Turing thought of first. Instead of examing one cell on a tape, the head of the 2-D Turing machine examines one cell on a grid.

We call a 2-D Turing machine tape head a "husk" in reference to its appearance on our display as a square outline around a grid cell. The name "husk" also refers to the fact that the tape head iself is only the the physical enbodiment, the outer shell, of the complicated and hidden Turing machine that may be controlling its behavior.

What if this Turing machine were not hidden? What if the invisible state-transition table for the machine was replaced by some sort of program physically present within the grid? In fact, what if the interpretation of this program to control the Turing machine was performed by yet another Turing machine scanning the program? That would be interesting. The behavior of that interpreter Turing machine could be controlled, in turn, by the actions of yet another Turing machine that interprets a controlling program. On a real computer, of course, there must be some level at which the Turing machines are controlled by a program that is not present on the grid. That is the role of our GRIP interpreter; it embodies the state-transition table for a Turing machine whose function is to interpret programs on the grid written in the GRIP language, which specify the behavior of other Turing machines on the grid.

Of course, normal Turing machines are not capable of directly controlling others, so we augment our machines' capabilities to include the creation of a "child" Turing machine, and a facility for controlling the child; for everything that a Turing machine can do, it can also cause its child to do that thing. This includes causing its child to create a child of its own, and control it in a certain way. Thus, for each Turing machine, there is really an infinite chain of descendant machines that it can create and control. In the *Grip* language, a top-level Turing machine can be told to make its descendants do something using a GRIP command of the form, "Tell your child to tell its child to ... tell its child to do *this*."

Our sc Grip programming system supports simultaneous execution of multiple top-level turing machines, or "roots" as we call them; thus one can write concurrent programs that cooperate or compete or attempt to destroy each other in a fashion reminiscent of "Core Wars."

2 Inspirations/History

3 Motivating Principles

4 Language Features

Every command in the GRIP language occupies a single grid cell, although the command may have immediate data values located in adjacent cells. The simplest commands are the flow-control instructions. These control the direction of motion of the root husk while it is executing the GRIP program. Figure 1 gives some examples.

In the examples above, the light squares represent the husks that are moving over the grid. In example 1, there are only directional flow-control instructions. The root husk will continue around the loop indefinitely.

The second example illustrates the fundamental "{" (BEGIN) command that tells the root husk to send a command to its child. If the child does not exist, it is created in the same position as its parent, namely, on the BEGIN instruction. The command sent is "L" (turn left), so the child rotates left from its current orientation (east) and the moves forward, or north. The "}", or END, is needed to stop controlling the child; this allows commands to the child to be strung together, as demonstrated in example 3.

Also in example 3 is an illustration of the "write" instruction, which takes a single argument.

Here is an example of a table.

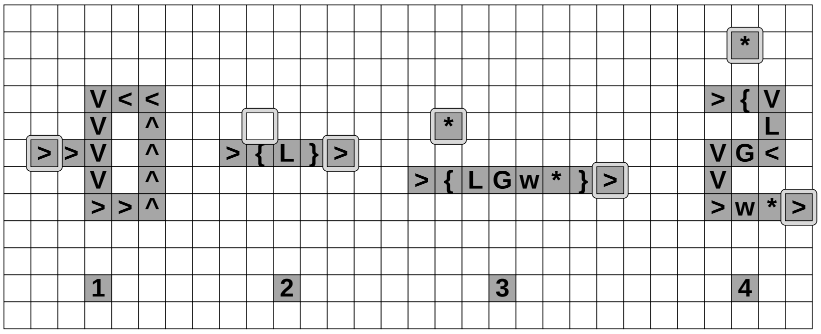

Figure 1: Some examples of GRIP commands.

Husk Control Primitives
Go forward 1 square
Turn left and go 1 square
Turn right and go 1 square
Turn around and go 1 square
Read symbol in current square
Write a symbol to current square
Create a child husk at current location
Apply a control primitive to the child husk
Destroy the child husk

Table 1: Husk Control Primitives.

Command Name	Command Format (for Eastward-moving root)	Description	Depends on child?
Root husk (program counter) direction control			
east	`>`	Go east	
south	`V`	Go south	
west	`<`	Go west	
north	`^`	Go north	
port	`P`	Turn left and go	
starboard	`S`	Turn right and go	
slash	`/`	Bounce E¡-¿S, W¡-¿N	
backslash	`\`	Bounce E¡-¿N, W¡-¿S	
bounce	`#`	Bounce back	
Interface to husk control primitives			
go	`G`	Go forward	Y
left	`L`	Turn left and go	Y
right	`R`	Turn right and go	Y
around	`A`	Turn around and go	Y
read	`r` s	Replace s with symbol being read	Y
write	`w` s	Write s	Y
begin	{	Start controlling child, create if needed	Y
end	}	Stop controlling child	Y
destroy	)	Stop controlling child and destroy it	Y
Branching			
if	`?` s x	Do x only if reading s	Y
if-not	`N` s x	Do x only if not reading s	Y
random	`~` x	Do x with probability 0.5	
case	`C` v_1 x_1 $\ldots v_n$ x_n □ `C`	If reading v_i, do x_i	Y
Miscellaneous			
define	`D` s	Define function for symbol x	
end-define	`E`	Return from function	
cross	`X`	No-op (allows paths to cross)	
skip	`$` x	Do not do x (skip over x)	
fork	`Y` x	Create new root starting at x	
suicide	`@`	Destroy root	

Table 2: Commands of the GRIP language.

5 Ease of Debugging

6 Implementation

References

[1] A. K. Dewdney. Two-dimensional turing machines and tur-mites make tracks on a plane. *Scientific American*, pages 180–183, September 1989.