# The Computational Completeness of Ray's Tierran Assembly Language[1]

Carlo C. Maley
MIT Artificial Intelligence Lab
545 Technology Sq.
Cambridge MA, 02139 USA
cmaley@ai.mit.edu

## 1 Introduction

One of the strengths of Ray's model, Tierra, is the open ended nature of evolution in the model. It is therefore important to show that Tierran evolution is not limited by computational incompleteness. Dr. Ray states that the assembly language of his model Tierra is computationally complete, due to similarity with other assembly languages that have been proven to be computationally complete.[2] Although this assertion may seem a reasonable at first, there is one important difference between the Tierran language and most other assembly languages: the lack of an instruction for reading input. I present a formal proof of the computational completeness of the Tierran language by showing how a Tierran can implement a Turing Machine. Thus, on closer examination, the idiosyncrasies of the Tierran language turn out to be theoretically insignificant, though perhaps practically crippling. I conclude by suggesting the implementation of an instruction for reading input. This minor expansion to the model has significant implications, including the possibility for self repairing genotypes and behaviour that is closely coupled to the organism's environment.

---

[1]This paper was published in Artificial Life III, C.G. Langton (ed.), Addison-Wesley (1994).
[2]See Ray (1991) p.372 with reference to Aho et al. (1974).

### 1.1 Computational Completeness

A computer language, along with the processor that executes it, is considered computationally complete if it is theoretically equivalent to a Universal Turing Machine.[3] The Turing Machine is a mathematical formalism that defines the theoretical limitations of computing. The computer is one instantiation of a machine that is theoretically equivalent to a Turing Machine. In other words, anything a computer can do, a Turing Machine can do, and vice versa. The Turing Machine is useful, not because someone would ever want to program it to do useful things, but because it is a rigourously defined mathematical construct which can be used to prove the power and limitations of computing.

### 1.2 The Suspicion

The basic model of computational execution is that a processor executes a program which takes some data as input and produces some data as output. In this way, a program can be seen as a function that maps a domain of input onto a range of output. Most computer languages have some instructions for reading input as well as instructions for writing output. However, the Tierran assembly language combines the two operations in the instruction MOV_IAB (short for "move instruction from address B to address A"). The MOV_IAB instruction simply copies the contents of memory address B into the contents of memory address A. Thus, reading from B and writing to A happen in a single instruction. It is therefore impossible to examine directly the contents of a memory address. This could pose serious problems for designing a program that would examine the input and produce one of a number of outputs depending on that input. Dr. Ray asserts that the Tierran language is computationally complete based on similarities with other assembly languages that have been proven to be computationally complete.[4] However, these other languages all have separate instructions for input and output, usually in the form of READ and WRITE instructions, and the proofs of their computational completeness depend on these separate instructions. The lack of independent input and output instructions in the Tierran language suggests the need for an argument to support the computational completeness of the the language that is not based on these somewhat dissimilar models.

## 2 Proving Computational Completeness

Perhaps the easiest way to prove that a computational system is computationally complete is to show that it is no more or less powerful, computationally, than a Turing Machine. A

---

[3]The Universal Turing Machine is a specific kind of Turing Machine that can emulate any other Turing Machine. It performs the emulation by working from an encoding of a specific Turing Machine which has been placed on the Universal Turing Machine's memory tape. In this way, a Universal Turing Machine is programmed to act like another form of Turing Machine. It follows from the fact that a Universal Turing Machine is a member of the class of Turing Machines, that if a Tierran can emulate any Turing Machine, it can emulate a Universal Turing Machine.

[4]Ray (1991) p.372 with reference to Aho et al. (1974).

constructive proof of computational completeness consists of two parts. First, it must be shown that a Universal Turing Machine can simulate the behaviour of the computational system in question. This establishes that a Turing Machine is at least as powerful as the computational system. Second, the computational system must be able to simulate the behaviour of Turing Machines, establishing the fact that the computational system is at least as powerful as a Turing Machine. If these two parts can be proven, the computational system must be equivalent to a Turing Machine.[5]

## 2.1 Turing Machine Emulation of a Tierran

The computational system in question here is a Tierran organism. The Tierran consists of two distinct parts, a virtual processor, and a program to be executed by that processor. In order to understand the power and limitations of a Tierran, I will specify the relevant aspects of both the virtual processor and the Tierran assembly language.

### 2.1.1 The Virtual Processor

The virtual processor of a Tierran consists of a number of registers and a stack. A register is a small amount of memory space that can be manipulated quickly. In a typical computer processor, the processor can only manipulate numbers that are held in registers. For example, in order to add two numbers, those numbers must first be loaded into two registers. After the numbers have been added, the sum is automatically placed in a register where it can then be either used again, or stored for future use in the normal memory space. In contrast, a stack is a larger form of temporary memory storage with the property that only the number most recently stored on the stack is immediately accessible.[6]

The Tierran virtual processor has two registers, Ax and Bx, for holding addresses of locations in the memory "soup." It also has two registers, Cx and Dx, for holding integers. In addition, the virtual processor has a stack which can hold up to ten numbers, either integers or addresses. Finally, it has a register called the "instruction pointer" (IP) which holds the address of the next assembly language instruction to be executed by the processor. For the purposes of the proof, I will be generous and ignore the finite limitations on stack size and memory space in Tierra.

---

[5]Alternatively, by Theorem 7.9 of Hopcroft and Ullman (1979), we need only prove that we can program a Tierran to simulate a two counter machine, but this is only slightly easier than simulating a Turing Machine for a Tierran. The only significant difference is that instead of Writing a symbol to the type, the Tierran would have to keep two counters. A counter can easily be simulated by incrementing or decrementing numbers held in the Tierran's registers.

[6]A stack is similar to a Pez candy or a dining hall plate dispenser. You can only see the candy on top, and if you decide that you want the third candy from the top, you must first remove, or "pop," the top two candies before having access to the desired candy. However, the dining hall plate dispenser is a better analogy because you can also "push" things onto the top of a stack, thus making the previous top plate inaccessible.

### 2.1.2 The Tierran Language

The Tierran language consists of 32 instructions, each a single word with no arguments. The lack of arguments is achieved by defining the instructions so that they will always operate on the contents of specific registers.[7] The description of each Tierran instruction follows. Note that the instructions in **bold** are the only ones used in the implementation of a Turing Machine.

**NOP_0**: "no operation 0" does not affect the state of the processor, acting as a template pattern[8] for the purpose of addressing locations in the program code.

**NOP_1**: "no operation 1" again does not affect the state of the processor, but can form binary patterns along with NOP_0.

**OR1**: "binary OR for the first bit" reverses the first (lowest order) bit of the integer in the Cx register.

**SHL**: "bitwise shift left" shifts the bits in the Cx register one digit to the left and introduces a 0 to the newly cleared lowest order bit, effectively doubling the number in the register. Note that a combination of OR1's and SHL's can construct any number in the Cx register.

**ZERO**: "zero" clears the number in the Cx register by setting it to zero.

**IF_CZ**: "if Cx is zero" is a conditional branch point instruction, which will only execute the following instruction if there is a zero in the Cx register. Otherwise, it will skip over the instruction that follows IF_CZ.

SUB_AB: "subtract Bx from Ax" finds the distance from the location stored in Ax to the location stored in Bx, and puts the result in the Cx register.

SUB_AC: "subtract Cx from Ax" adjusts the address in Ax backwards by the number of units held in the Cx register.

INC_A: "increment Ax" moves the address in Ax one unit forward.

INC_B: "increment Bx" moves the address in Bx one unit forward.

**INC_C**: "increment Cx" adds 1 to the number in Cx.

**DEC_C**: "decrement Cx" subtracts 1 from the number in Cx (zero is the lower bound).

**PUSH_AX**: "push Ax onto the stack" puts the address in Ax on the stack.

**PUSH_BX**: "push Bx onto the stack" puts the address in Bx on the stack.

**PUSH_CX**: "push Cx onto the stack" puts the number in Cx on the stack.

**PUSH_DX**: "push Dx onto the stack" puts the number in Dx on the stack.

**POP_AX**: "pop the stack into Ax" removes the number on top of the stack and stores it in Ax.

**POP_BX**: "pop the stack into Bx" removes the number on top of the stack and stores it in Bx.

**POP_CX**: "pop the stack into Cx" removes the number on top of the stack and stores it in Cx.

---

[7] In effect, the lack of arguments makes the Tierran language closer to a machine code than an assembly language.

[8] "Addressing by template" is a new form of addressing invented by Dr. Ray. Where a traditional assembly language would use a label to indicate a location in the code, a Tierran uses a string of NOP_0's and NOP_1's as a binary pattern to mark the location of the instruction directly following the pattern. A location is requested by giving the complement of the desired pattern. This is similar to the way in which proteins use templates on their surfaces to locate other proteins. See Ray (1991) for a more detailed explanation of addressing by template.

**POP_DX**: "pop the stack into Dx" removes the number on top of the stack and stores it in Dx.

**JMP**: "jump" forces execution to jump to the address specified by the template pattern that follows the JMP instruction by loading that address into the IP register.

JMPB: "jump backwards" works the same way as JMP except the address must be behind the location of the JMPB instruction.

**CALL**: "call a procedure" stores the current location of the **IP**, as an offset from the starting location of the program, on the stack and then jumps the execution to the location specified by the template pattern following the CALL instruction.

**RET**: "return from a procedure" returns execution to the location stored on the top of the stack, which should be the location following the last CALL instruction to be executed.

**MOV_CD**: "move the contents of Cx into Dx" copies the contents of Cx and stores the number in Dx.

**MOV_AB**: "move the contents of Ax into Bx" copies the contents of Ax and stores the address in Bx.

**MOV_IAB**: "move the instruction at location Bx to location Ax" copies the instruction at the location held in Bx, and writes it into the location held in Ax.

ADR: "find the address" stores the address of the location specified by the template pattern following the ADR instruction, in the Ax register.

**ADRB**: "find the address behind" acts the same as ADR except that the specified location must be behind the ADRB instruction.

ADRF: "find the address in front" acts the same as ADR except that the specified location must be in front of the ADRF instruction.

MAL: "allocate memory" allocates a continuous block of unused memory the size of the number held in Cx. The address of the start of this block is stored in the Ax register.

DIVIDE: "divide the cell" introduces a new Tierran to the system, and removes the "mother's" ability to alter the "daughter's" program code.

It should be noted that a Tierran may read and execute any instruction found in the soup, but can only write into its own genome and the genome of its daughter before "cell division". These restrictions on the ability to alter the data in the soup are thought to be analogous to the protection afforded by a cell wall.

To show that a Turing Machine is at least as powerful as a Tierran, I must show that a Universal Turing Machine can simulate the behaviour of a Tierran. This has been done separately by both Dr. Ray and myself, since we have programmed a computer, which is a Universal Turing Machine, to run the model Tierra. Thus, the formal proof is the program code of the Tierra model.[9]

## 2.2 Tierran Emulation of a Turing Machine

[9]The code for the Tierra model can be copied via anonymous FTP from the node: life.shls.udel.edu

The real question under examination is whether a Tierran is as powerful as a Universal Turing Machine. To prove this constructively, I must show that a Tierran can be programmed to be a Turing Machine.

### 2.2.1 Definition of a Turing Machine

A Turing Machine (**M**) can be denoted by a 7-tuple, $\mathbf{M} = (\mathbf{Q}, \Sigma, \mathbf{T}, f, \mathbf{q_0}, \mathbf{b}, \mathbf{F})$ where[10]:

> **Q** is the set of possible *states*,
> **T** is the finite set of allowable *tape symbols*,
> **b**, a symbol in **T**, is the *blank* symbol,
> $\Sigma$, a subset of **T** not including **b**, is the set of *input symbols*,
> $f$ is the *next move function*, a mapping from **Q** x **T** to **Q** x **T** x {L, R}
> ($f$ may be undefined for some arguments),
> $\mathbf{q_0}$, a state in **Q** is the *start state*,
> and **F**, a subset of **Q**, is the set of *final states*.

The Turing Machine thus has a finite set of states **Q**. It is conceived of as a machine, which at any given time is "in" one of the states of **Q** (see figure 1). The Machine begins processing in the state $\mathbf{q_0}$. After reading a symbol on an infinite one dimensional tape, it may change state, write a symbol on the tape and move the machine head that reads from and writes on the tape either left (**L**) or right (**R**), according to the function $f$.[11] For a thorough treatment of the theory of Turing Machines, see Hopcroft and Ullman (1979).

---

[10] This definition has been taken directly from Hopcroft and Ullman (1979) p. 148.

[11] The Turing Machine can be said to have accepted the input on the tape if it halts, after reading the input, in one of the states in **F**. One way of using a Turing Machine is to distinguish between different strings of input symbols on the tape by either accepting or rejecting the strings. Since a Tierran never halts, "halting" could be simulated by sending the Tierran into an infinite loop when it finishes processing the input symbols.

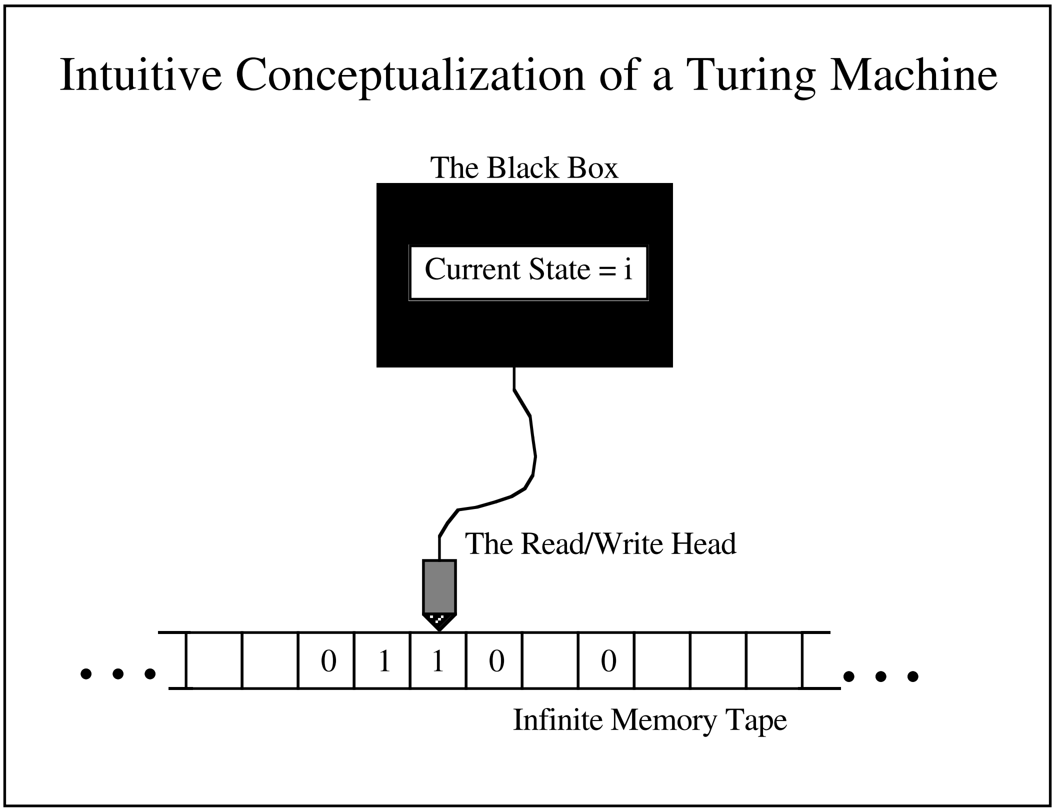


**Figure 1**

### 2.2.2 Construction of a Turing Machine

To construct a Turing Machine, I will show how a Tierran can simulate each part of the Turing Machine. A Tierran genotype is a series of adjacent instructions. In this case the Tierran's genotype will contain both the program for simulating the Turing Machine and the tape of input symbols. The two possible input symbols on the tape are ZERO and INC_C. Before reading a type symbol the Cx register is initialized with a 1, so that after an input symbol is read the contents of Cx will be either 0, 1, or 2, depending on whether a ZERO, *blank* , or an INC_C was read. So $\Sigma = \{0, 2\}$, effectively a binary code, as in a traditional computer. Let the *blank* symbol **b** = NOP_0, so that **T** = {ZERO, INC_C, NOP_0}. The read/write head of the Turing Machine can be simulated with the **IP** of the Tierran. After every input symbol there is a CALL to the procedure which implements the ***f*** function. For example:

```
NOP_0        : an input symbol.
CALL
(Template pattern for the f function.)
ZERO         : the second input symbol.
CALL
...
```

Let $\mathbf{Q} = \{\mathbf{q_0}, \mathbf{q_1}, ..., \mathbf{q_n}\}$. The current state of the Turing Machine is indicated by the contents of Cx, which holds the integer **i**, if the Turing Machine is in state $\mathbf{q_i}$. Thus, $\mathbf{q_0}$ is simulated by holding a zero in the Cx register. **F** is simply a set of integers. The register Cx can be used for both reading input and for storing the current state by using the Stack and the Dx register to hold one of the two numbers in temporary storage. The following algorithm might be used to swap two numbers, n1 and n2, when n1 is in the Cx register and n2 is in the Dx register:

<u>Swap Registers:</u>

```
Starting State    : Cx = n1, Dx = n2, Stack = empty.
PUSH_DX           : Cx = n1, Dx = n2, Stack = n2.
MOV_CD            : Cx = n1, Dx = n1, Stack = n2.
POP_CX            : Cx = n2, Dx = n1, Stack = empty.
```

In this way, a Tierran Turing Machine might juggle both the input number and the current state number without loosing any information, as long as the stack is not full.

Note that Cx is the register that is most easily manipulated, both for loading numbers, via the ZERO, OR1, SHL, DEC_C, and INC_C, and for branching using the IF_CZ instruction. To call one of a variety of procedures, depending on the value of some variable, the Tierran must load the number into the Cx register, then keep decrementing it using DEC_C and testing if it is zero, using the IF_CZ.

<u>Call Procedure n:</u>

```
Current State     : Cx = n.
IF_CZ
CALL
(Template pattern for procedure if n = 0.)
DEC_C             : Cx = n - 1.
IF_CZ
CALL
(Template pattern for procedure if n = 1.)
DEC_C             : Cx = n - 2.
...
IF_CZ             : Cx = n - i.
CALL
(Template pattern for procedure if n = i)
DEC_C             : Cx = n - (i + 1).
and so on.
```

This is the algorithm that would be used when the Turing Machine needs to determine its current state as well as when the Turing Machine determines whether it read a ZERO, INC_C, or NOP_O as input.

The only remaining aspect of the Turing Machine that needs to be simulated is ***f***. This is really the heart of a Turing Machine. I now have the algorithmic tools to determine the current state and the current input. These are the two arguments to ***f***. It is then trivial to use the two arguments to call a procedure ***f*****(n, t)**. This procedure must be able to change the current state, write a symbol on the tape, and move the "read/write head" either left or right one input symbol[12] on the tape. I will give an algorithm to deal with each of these challenges.

### 2.2.2.1 CHANGING STATE

Changing the current state is relatively trivial. Any number can be constructed, within the limits of the register by first clearing the Cx register with a ZERO instruction and then using OR1 and SHL to construct the bits of the number.

### 2.2.2.2 WRITING A SYMBOL

Writing a symbol to the tape can be done by calling one of three procedures: "write ZERO," "write INC_C," or "write NOP_0." All three procedures would have a similar structure. Note that the current position of the write head is on top of the stack.

<u>Writing a Symbol</u>:

```
(Template pattern for the symbol to be written.)
INC_A        : This is a dummy instruction that will never get
               executed.  It is necessary to separate the template
               pattern from the symbol to be written, which may be
               NOP_0.
(Symbol to be written)  : either ZERO, INC_C, or NOP_0.
(Template pattern identifying the procedure)
               : The starting point for the execution of the
               procedure.
ADRB
(Template pattern for symbol to be written)
               : Loads Ax with address of the above INC_A.
INC_A          : Ax = the address of the "Symbol to be written."
MOV_AB         : Ax = Bx = location of symbol to be written.
```

[12]Note that the tape has CALL and pattern instructions between the input symbols, so moving the read/write head one input symbol in either direction means skipping over those instructions.

```
POP_AX      : Ax = tape head location, Bx = location of symbol to be
              written.
PUSH_AX     : Stack = tape head position, Ax = tape head position,
              Bx = location of symbol to be written.
MOV_IAB     : finally writes the symbol to the tape.
```

2.2.2.3 MOVE THE READ/WRITE HEAD

The final challenge is to implement the movement of the read/write head of the Turing Machine. The read/write head was the **IP** before the CALL instruction was executed. Since an address is just an integer[13] it can be manipulated in the Cx register. The algorithm for moving the read/write head one input symbol to the left follows. Again, the current position of the read/write head has been placed on the top of the stack by the CALL to the ***f*** procedure. Note that moving one input symbol to the right requires only that the DEC_C instruction be replaced by the INC_C instruction.

<u>Move Read/Write Head Left:</u>

```
Current State:    Cx = n, Stack = r/w head.
POP_BX      : Bx = r/w head, Cx = n, Stack = empty.
PUSH_CX     : Bx = r/w head, Cx = n, Stack = n.
PUSH_BX     : Bx = r/w head, Cx = n, Stack = r/w head, n.
POP_CX      : Bx = r/w head, Cx = r/w head, Stack = n.
DEC_C       : Bx = r/w head, Cx = r/w head - 1, Stack = n.
(DEC_C will have to be repeated m times in order to skip over the CALL
and pattern template instructions that separate the input symbols.)
            : Bx = r/w head, Cx = r/w head - m, Stack = n.
PUSH_CX     : Bx = r/w head, Cx = r/w head - m,
                  Stack = r/w head - m, n.
POP_BX      : Bx = r/w head - m, Cx = r/w head - m, Stack = n.
POP_CX      : Bx = r/w head - m, Cx = n, Stack = empty.
PUSH_BX     : Cx = n, Stack = r/w head - m.
(Then I prime Cx for reading the next input symbol, and return to
reading the input.)
MOVCD       : Cx = n, Dx = n, Stack = r/w head - m.
ZERO        : Cx = 0, Dx = n, Stack = r/w head - m.
INC_C       : Cx = 1, Dx = n, Stack = r/w head - m.
RET         : Cx = 1, Dx = n, Stack = empty,
```

---

[13]In my version of Tierra, called Terra, an address indicates a position in two-dimensional space and so is a pair of integers. The addition of "wrap-around" borders makes the algorithm for moving the read/write head left or right significantly more complicated, but still theoretically possible.

```
IP = new r/w head position = the old r/w head - m.
```

I now have algorithms for reading input, calculating the current state and branching to the appropriate procedure for executing $\boldsymbol{f}(\mathbf{n}, \mathbf{t})$. The procedure $\boldsymbol{f}(\mathbf{n}, \mathbf{t})$ itself is made up of three algorithms, calculating a new state, writing a symbol to the tape, and moving the read/write head. Combining all of these algorithms is a trivial if tedious exercise resulting in a fully functional Turing Machine. Thus, a Tierran is theoretically as powerful as a Turing Machine and so is also computationally complete.

## 3 Limitations of Tierrans

It should be clear by now that although theoretically complete, the Tierran's behaviours are highly limited by the nature of the instruction set. At present, examining the contents of a memory cell can only be accomplished by executing the instruction in that cell and then examining the change in the Tierran's state. This only works well for instructions that have easily determined effects on the Tierran's state, like ZERO. This is how I implemented the reading of input symbols in the Tierran Turing Machine. Ray has given Tierrans freedom to read (in the sense that they may find locations using the ADR family of instructions) and copy the instructions of any other Tierran in the Tierran primordial soup. However, the freedom to examine instructions is of little use when the Tierran can only determine the memory contents indirectly. It would be a rather simple addition to break the MOV_IAB instruction into a READ and a WRITE instruction.

Furthermore, Tierrans are restricted to only writing into their own genome and their daughter's genome prior to "cell division." This is analogous to the protection afforded by cell walls, and prevents the Tierrans from killing each other off in the same way as their conceptual ancestors, the Core Warriors, eradicate each other.[14] Yet, without a READ instruction, writing into one's own genome is largely a blind and suicidal exercise. With a READ instruction, self-modification could be a practical and perhaps an adaptive exercise for a Tierran.

As a final example of the power of having separate READ and WRITE instructions, there follows the outline of an algorithm for error checking reproduction. Like the analogous DNA repair mechanisms, such a program would severely diminish the rate of mutations due to copying errors in reproduction.

```
READ parental gene.
WRITE it into the daughter's genome.
READ the daughter's gene that was just written.
Compare it to the parental gene.
```

[14]Reference Dewdney 1984, 1985, 1987, and 1989 for information on Core Wars.

`If they are different, loop to the top and try again.`

## 4 Conclusion

I have shown that the Tierran language is computationally complete. This means that Tierrans may theoretically evolve to perform any task a digital computer can perform. However, this is clearly only a theoretical possibility. In practice, a Tierran is severely limited by the nature of the Tierran language. By splitting the MOV_IAB instruction into a READ and a WRITE instruction, Tierrans would gain the possibility of examining and reacting intelligently to their environment.

## Bibliography


Aho, A.V., Hopcroft, J.E., and Ullman, J.D. (1974) The Design and Analysis of Computer Algorithms. Addison-Wesley: Reading, MA.

Dewdney, A.K. (1984) "Computer Recreations: In the Game Called Core Wars Hostile Programs Engage in a Battle of Bits." *Scientific American* **250**, May: 15-19.

Dewdney, A.K. (1985) "Computer Recreations: A Core War Bestiary of Viruses, Worms and Other Threats to Computer Memories." *Scientific American* **252**, March: 14-19.

Dewdney, A.K. (1987) "Computer Recreations: A Program Called MICE Nibbles Its Way to Victory at the First Core War Tournament." *Scientific American* **256**, January: 8-11.

Dewdney, A.K. (1989) "Computer Recreations: Of Worms, Viruses, and Core War." *Scientific American* **260**, March: 90-93.

Hopcroft, J.E., and Ullman, J.D. (1979) Introduction to Automata Theory, Languages, and Computation. Addison-Wesley: Reading MA.

Ray, T.S. (1991) "An Approach to the Synthesis of Life" in Artificial Life II. Addison-Wesley: Reading MA. pp. 371-408.