CoreWar

Ein Spiel nicht nur für Programmierer

Inhalt

1. Einleitung
2. Geschichte von CoreWar
3. Die Spielidee von CoreWar
4. Redcode-Programme
5. Der Redcode-Compiler
6. Der Wettkampf
7. Die Spieloberfläche von CoreWar
8. Für Einsteiger: der Assistent
9. Ausblicke
10. Die Programmierschnittstelle
11. Autor & Copyright

Anhang A: Eine Einführung ins Programmieren am Beispiel von Redcode

1. Einleitung

CoreWar ist ein Spiel, das man nicht nur mit dem Computer spielt, sondern das auch IN einem Computer spielt. In CoreWar kämpfen zwei von Menschen programmierte Programme in einem "zyklischen" Speicherbereich. Ziel des Spieles ist es, das gegnerische Programm handlungsunfähig zu machen. Ein Programm verliert, wenn es auf eine Anweisung trifft, die es nicht ausführen kann.

2. Geschichte von CoreWar

Die Spielidee "CoreWar" wurde von A. K. Dewdney im Artikel "Computer Recreations" der Zeitschrift "Scientific American", Ausg. Mai 1984, Seite 15 ff. erstmals beschrieben.
Die erste Implementierung wurde von Kevin Byorke am 28. Mai 1994 fertiggestellt und zum Kopieren für nichtkommerzielle Zwecke freigegeben.
R.Green und R.Sayer konvertierten CoreWar nach MS-DOS. Die Version 1.01 vom 4. August 1985 ist dann in meine Hände gelangt und es regte sich der Wunsch in mir, eine zeitgemäße Version (in Windows) zu programmieren.

Der Name CoreWar stammt von einer - inzwischen veralteten - Möglichkeit, Arbeitsspeicher physikalisch herzustellen. Heute (1995) besteht der Arbeitsspeicher eines Computers aus RAM, Random Access Memory, kleinen Chips aus Silizium. Als in den 50'er und 60'er Jahren die Halbleitertechnologie noch nicht so weit fortgeschritten war, wurden Ferritkerne (engl.: core), meist in Form von Ringen, die in einem Drahtnetz befestigt sind) als Speicher verwendet. Jeder Kern konnte ein Bit (eine Ja/Nein-Entscheidung) speichern, je nach Magnetisierung. CoreWar spielt in einem solchen Speicher eines (virtuellen) Computers, daher der Name CoreWar.

3. Die Spielidee von CoreWar

Zwei Computerprogramme kämpfen in einem abgegrenzten Speicherbereich darum, das gegnerischen Programm zum Absturz zu bringen. Während dem eigentlichen Kampf bleibt der (menschliche) Spieler in der Beobachterrolle. Er wird im Vorfeld aktiv, wenn es darum geht, ein möglichst gutes, d.h. robustes und angriffstarkes Programm fertigzustellen, das er an den Start schicken kann. Seine Hauptaufgabe liegt darin, ein solches CoreWar-Kampf-Programm zu programmieren. Ein schneller Daumen am Abzug einer Laserpistole oder

schnelles Reaktionsvermögen ist nicht gefragt.

CoreWar-Programme sind in einer speziellen, assemblerähnlichen Programmiersprache geschrieben, die Redcode genannt wird. Redcode besitzt gerade neun verschiedene Befehle. In Gegensatz zu normalen Assembler-Code belegt jede Redcode-Anweisung inklusive der zu der Anweisung gehörenden Argumente genau eine Speicherstelle. Damit ist die Speicher-Adressierung relativ einfach. Z.B. bedeutet die Anweisung JMP -2 daß zwei Anweisungen zurückgesprungen wird; JMP 0 ergibt eine Endlosschleife, und JMP 1 bewirkt einfach nur einen Sprung zur nächsten Speicherstelle - d.h. JMP 1 entspricht der “No-Operation”-Anweisung.

Ein Programm weiß weder, wo es sich im Speicher befindet, noch wo das gegnerische Programm liegt. Der Speicher von CoreWar ist zyklisch, d.h. wenn ein Programm auf der einen Seite aus dem Speicher herausläuft, taucht es automatisch an der anderen Seite wieder auf. Alle Adressen sind daher relative Adressen, absolute Speicheradressen sind überflüssig.

Die neun Redcode-Anweisungen (Opcodes) sind:

Opcode	Argumente		Beschreibung/Ergebnisse
MOV	A	B	A wird nach B kopiert (A bleibt unverändert)
ADD	A	B	B wird zu B + A (A bleibt unverändert)
SUB	A	B	B wird zu B - A (A bleibt unverändert)
JMP	A		Programm springt zu A
JMZ	A	B	Programm springt zu A wenn B = 0
JMG	A	B	Programm springt zu A wenn B > 0
DJZ	A	B	B wird zu B-1; ist das Ergebnis Null, springt das Programm zu A
CMP	A	B	Die nächste Anweisung wird übergangen, außer wenn A = B
DAT		B	B sind “reine” Daten. Ein Programm verliert, wenn es versucht, eine DAT-Anweisung auszuführen.

Es gibt außerdem drei Adressierungs-Modi: Direkt, Indirekt und Unmittelbar. Adressierungs-Modi sind die verschiedenen Arten, wie ein Argument (ein Operand) interpretiert werden kann.

Im Direct-Mode bezeichnet das Argument eine (relative) Adresse einer Speicherstelle, welche die Daten enthält. Dies ist der voreingestellte Modus. So würde die Anweisung `DJZ 12 -1` die vorige Speicherstelle um eins erniedrigen. Ist diese daraufhin Null, wird um 12 Speicherstellen nach vorne gesprungen (wenn nicht, würde ganz normal die auf die Anweisung folgende Anweisung ausgeführt werden).

Im Indirect-Mode, der durch einen Klammeraffen "@" gekennzeichnet wird, bezeichnet das Argument eine Speicherstelle, die einen Zeige auf die gewünschten Daten enthält.
Beispiel:

```
JMP     @1
DAT     1
JMP     12
CMP     -10     -11
```

Die erste Sprunganweisung schaut nach dem Sprungziel in der relativen Adresse +1. In diesem Beispiel befindet sich dort eine 1, was bedeutet, daß der Sprung relativ zu dieser Adresse 1 ist - es wird also zur Anweisung `JMP 12` gesprungen. Wenn es anstelle von `DAT 1` gelautet hätte `DAT 2`, so wäre zum `CMP`-Befehl gesprungen worden.
`DAT -1` würde zum ursprünglichen Sprungbefehl zeigen und das Programm würde in einer Endlosschleife hängen (unter der Voraussetzung, daß das feindliche Programm nicht dazwischenfunkt). Bei `DAT 0` würde das Programm zur `DAT`-Anweisung springen und mit einem Fehler abbrechen (leider verloren...).

Schließlich gibt es noch den Immediate-Modus: Das Argument wird als eine absolute Zahl genommen; kenntlich gemacht wird dies durch ein vorangestelltes Nummernzeichen "#".
`CMP #2 -3` überprüft, ob der Wert an der Speicherstelle -3 identisch mit der Zahl 2 ist.
Ein `MOV`-Befehl mit einem Immediate-Argument als erstes Argument bewirkt, daß die im zweiten Argument angegebene Speicherstelle eine `DAT`-Anweisung wird. Dies ist ein einfacher Weg, um "Bomben" in das feindliche Programm zu schießen. Wäre das erste Argument kein Immediate-Mode, so wären Befehl und Argument kopiert worden - mit dieser Methode kann sich ein Programm im Speicher fortbewegen.

Welches Argument ist welches?
Das zweite, auch "B" genannte Argument, ist das, welches die Daten für `ADD`, `SUB`, `CMP` und die bedingten Sprünge beinhaltet. Wenn z.B. die Redcode-Sequenz

```
ADD     #2      1
CMP     3       15
```

ausgeführt wird, wird die zweite Zeile durch den `ADD`-Befehl geändert, so daß der `CMP`-Befehl jetzt die Speicherstellen 3 und 17 vergleicht. Das erste "A" Argument wird von Redcode nie geändert.

Beide Kampf-Programme werden durch das CoreWare-Betriebssystem MARS ausgeführt. Der MARS-Interpreter schaltet abwechselnd zwischen den beiden Programmen hin und her und gibt jedem Programm einen Takt Rechenzeit, dann ist das andere Programm dran. Das geht so lange, bis ein Programm verliert, eine bestimmte Anzahl von Anweisungen ausgeführt wurde (als Sicherung gegen Endlosschleifen) oder der Benutzer die Programmausführung abbricht.

4. Redcode-Programme

Das kürzest mögliche Kampf-Programm ist "IMP"

```
MOV     0       1
```

IMP kopiert sich selbst an die nächste Speicherstelle, schreitet zur nächsten Speicherstelle fort, kopiert sich wieder ein Feld weiter etc. Obwohl das ursprüngliche Programm kurz ist, kann es möglicherweise den gesamten Speicher füllen, wenn es nicht aufgehalten wird, und kann so das größtmögliche Kampf-Programm werden. Es kann auch seinen Gegner verwirren, denn jedes Programm, daß zu einer Speicherstelle springt, die von IMP beschrieben wurde, wird ein identischer Clone von IMP werden. Allerdings kann IMP auch (fast) nie eine Schlacht gewinnen, da es den Gegner nicht handlungsunfähig machen kann.

Hier ist ein anderes Programm, “ANTI-IMP”

```
MOV     #0      -5
CMP     #0      -6
JMP     -1
MOV     #0      -5
MOV     #0      -6
MOV     #0      -7
MOV     #0      -8
JMP     -7
```

ANTI-IMP setzt eine Markierung an der Stelle -5 relativ zum ersten Byte, dann waret es, bis IMP daherkommt. Wenn der Marker verändert wird, bombardiert es den Bereich, in den IMP soeben hineingelaufen ist, mit `DAT 0`, welches IMP nicht ausführen kann und daher ins Gras beißt.

Hier ist “ANTI-ANTI-IMP”

```
MOV      4    @3
ADD     #1     2
JMP     -2
DAT      2
MOV      0     1
```

ANTI-ANTI-IMP beschreibt einen Block mit Code, der wie IMP aussieht. Wenn ANTI-IMP diesen falschen Hasen bemerkt, wird er ihn angreifen, aber nichts erreichen - es wird einfach überschrieben werden und ein Clone von IMP werden (es rennt im Speicher herum und ist jetzt eine Gefahr für ANTI-ANTI-IMP, da es gegen IMP nicht gesichert ist.

Ein anderes Beispiel ist “DWARF”

```
JMP      2
DAT     -1
ADD     #5    -1
MOV     #0   @-2
JMP     -2
```

Dieses Programm feuert “Bomben”, ähnlich wie ANTI-IMP. DWARF ist aber nicht auf IMP spezialisiert, sondern trifft jedes Programm, welches größer als 5 Bytes ist (sofern DWARF nicht schon vorher selber getroffen wurde). Man beachte, daß bei einem Feld von Turnier-Größe (8000 Bytes) DWARF selber gegen seine eigenen Schüsse immun ist.

IMP und DWARF sind Vertreter einer Klasse von Programmen, die klein und sehr aggressiv sind, aber nicht intelligent sind.
In einem nächsten Schritt kann man Programme entwickeln, die zwar größer und weniger angriffslustig sind, aber intelligent genug sind, um sich gegen o.g. Kämpfer zu wehren.

Ein solches Programm ist z.B. “GEMINI”

```
JMP      3
DAT      0
DAT     99
MOV    @-2   @-1
CMP     -3    #9
JMP      4
ADD     #1    -5
ADD     #1    -5
JMP     -5
MOV    #99    93
```

GEMINI ist noch kein vollständiges Kampf-Programm. Seine einzige Funktion ist, sich selbst 100 Bytes vor die aktuelle Position zu kopieren und dann zu dieser neuen Kopie zu springen. Aber der GEMINI-Algorithmus läßt sich gut in Kampf-Programmen verwenden. So z.B. in JUGGERNAUT, welches sich 10 Bytes vorwärtsbewegt anstelle von 100. Wenn ein gegnerisches Programm von Teilen von GEMINI oder JUGGERNAUT überschrieben wird, ist die Wahrscheinlichkeit größer, daß es sich total daneben benimmt und verliert, als wenn es von IMP überschrieben wird (dann wird es nur eine Kopie von IMP und läuft bis in alle Unendlichkeit im Speicher rum).

Noch intelligentere Programme sind zu groß, um sich zu reproduzieren. Eines dieser Programme, welches von den Programmierern "RAIDER" genannt wird, hat z.B. zwei Vorposten um sich herum aufgestellt (Ähnlich ANTI-IMP)..

5. Der Redcode-Compiler

In der vorliegenden Version liegen der Recode-Compiler und die Spielumgebung als zwei getrennte Programme vor. Der Redcode-Compiler ist ein ganz gewöhnlicher Text-Editor (ähnlich Notepad), der zusätzlich die Möglichkeit hat, den Text zu compilieren. Für Anfänger gibt es noch einen Assistenten, der das Erstellen von Redcode-Programmen vereinfacht.

Bei der Eingabe sind folgende Regeln zu beachten

* In jeder Zeile darf nur eine Anweisung stehen.
* Die Groß/Kleinschreibung ist egal.
* Der Befehl und die Argumente müssen durch Leerzeichen oder Tabulator voneinander getrennt sein.
* Ungültige Adressierungsarten wie z.B. `MOV 1 #5` oder `JMP #1` oder `DAT @1` erzeugen bereits beim Compilieren einen Fehler.
* Kommentare werden durch ein Semikolon ";" vom Rest der Zeile abgetrennt. Auch reine Kommentarzeilen sind möglich.
* Jedes Programm kann Attribute besitzen. Attribute haben die Syntax "#Name Inhalt". Zur Zeit werden von der Spielumgebung nur die Attribute "NAME" und "(C)" ausgewertet.
* Attribute dürfen an jeder beliebigen Stelle im Quelltext erscheinen.

Ein für den Compiler gültiges Redcode-Programm sieht dann z.B. folgendermaßen aus:

```
#Name Dwarf
;Dwarf ist Englisch und bedeutet "Zwerg"
#(C) Demo-Programm
jmp      2          ; Das Programm startet immer am Anfang
dat     -5          ; Datenbereich muß übersprungen werden
add     #5     -1   ; Schleifenzähler erhöhen
mov     #0    @-2   ; "Bombe" verschießen
jmp     -2          ; Nächste Bombe...
```

6. Der Wettkampf

A. K. Dewdney hat für den Wettkampf folgende vier Regeln aufgestellt (aus dem engl.):

1. Die beiden Kampf-Programme werden an zufällige Adressen in den Speicher geladen, aber so, daß sie am Anfang mindestens 1000 Speicherzellen voneinander entfernt sind.
2. Es wird abwechselnd eine Anweisung von jedem Programm ausgeführt, bis ein Programm auf eine Anweisung trifft, die es nicht ausführen kann. Dieses Programm wird dann zum Verlierer erklärt.
3. Die Programme können sich mit jeder verfügbaren Waffe bekriegen. Eine "Bombe" kann eine 0 sein oder irgend eine andere Zahl; auch ein Redcode-Befehl ist als Bombe möglich.
4. Für jeden Wettkampf wird ein Zeitlimit festgesetzt, abhängig von der Geschwindigkeit des Computers. Wenn das Limit erreicht wird und beide Programme arbeiten noch, endet der Wettkampf unentschieden.

Nicht in den Regeln enthalten ist, daß Dewdney von einer Feldgröße von 8000 Speicherstellen ausgeht.

In der vorliegenden Implementation von CoreWar gibt es folgende Besonderheiten:
1. Jedes Redcode-Programm wird immer am Anfang gestartet. Deshalb muß bei Programmen aus der Literatur, die ihren Datenbereich am Anfang haben, als erstes ein `JMP` auf den eigentlichen Programmcode stehen.
2. Es wird kein Zeitlimit festgelegt, sondern eine maximale Zahl von Programmschritten.
3. Die Feldgröße und der relative Abstand der beiden Programme beim Start können vom Benutzer verändert werden.

7. Die Spieloberfläche von CoreWar

1. Die Kontrahenten
Die beiden Programme werden durch unterschiedliche Farben gekennzeichnet. Das obere Programm bekommt die Farbe Rot, das untere ist das blaue Programm.
Mit den Buttons "Programm Rot festlegen" und "Programm Blau festlegen" können die Programme ausgewählt werden, die gegeneinander antreten sollen. Nach den Auswählen werden Namen und Urheber der Programme in einem farbigen Kasten angezeigt.

2. Das Spielfeld
Das Spielfeld nimmt den größten Teil des Fensters ein. Jedes Kästchen entspricht einer Speicherstelle. Je nach Inhalt der Speicherstelle werden diese mit unterschiedlichen Symbolen gekennzeichnet:
* Ein leeres Feld bedeutet, daß noch kein Programm etwas in diese Speicherstelle hineingeschrieben hat.
* Ein Strich bedeutet, daß ein Programm an dieser Stelle Daten abgelegt hat (dies kann auch eine "Bombe" sein.
* Ein Kugel (in der Verkleinerung ein Quadrat) bedeutet, daß hier ein ausführbarer Befehl lagert.
* Ein dunkles Rechteck kennzeichnet den aktuellen Stand des Instruction Pointers (IP), d. h die Anweisung, die als nächste ausgeführt wird, wenn das Programm an die Reihe kommt.
* Ein schwarzes Rechteck kennzeichnet ein Feld, das außerhalb des Speichers liegt und von den Programmen nie erreicht werden kann.

Die Striche und Kugeln erscheinen immer in der Farbe des Programms, welches als letztes auf diese Speicherstelle schreibend zugegriffen hat.
Ein grünes Feld (Kugel oder Strich) bedeutet, daß Sie als User den Inhalt dieser Speicherstelle manuell während dem Spiel verändert haben.

Der “Besitz” einer Speicherstelle ist für den Programmablauf jedoch vollkommen unerheblich; ein Programm kann nicht dadurch gewinnen, daß es möglichst viele Speicherstellen besitzt. Ein Programm kann auch nicht dadurch verlieren, daß es auf eine Speicherstelle trifft, die dem feindlichen Programm gehört. Es verliert nur dadurch, daß es bei der Programmausführung auf eine Speicherstelle mit Daten stößt, die es nicht ausführen kann.

3. Die Buttons (Schalter)
Mit dem Button “*Kampf Start / Stop*” kann jederzeit der Kampf gestartet und wieder angehalten werden (Pausetaste).
Mit “*Rücksetzten*” werden die Kampf-Programme neu geladen und der Speicher zurückgesetzt. Dabei hat der Benutzer die Wahl, ob er den gleichen Kampf noch einmal ablaufen lassen möchte, oder ob der Zufallsgenerator (falls eingeschaltet) neue Ausgangs-Positionen bestimmen soll.
Die Buttons “*Programm Rot festlegen*” und “*Programm Blau festlegen*” führen zu Dialogen, in denen der Benutzer festlegen kann, welche Programme gegeneinander antreten sollen.
Mit dem Button “*Optionen*” können Sie die Kampf-Parameter in weiten Grenzen selber einstellen. Der Button “*CoreWar beenden*” bedarf wohl keiner weiteren Erläuterung.

4. Die Optionen

Größe des Spielfeldes
Zunächst können Sie die Größe des Spielfeldes wählen. Von Dewdney werden 8000 Bytes vorgeschlagen (“Turnier-Größe”). Im Regelfall hat der Speicher jedoch nur eine Größe von 1024 Bytes. Die Spielfeldgröße kann auch durch Zufall oder durch direkte Eingabe der Größe festgelegt werden.

Verzögerung
Der Interpreter muß in regelmäßigen Abständen Pausen einlegen, damit in der Zwischenzeit die Windows-typischen Routinen weiterlaufen können. Man kann festlegen, nach wie vielen Programmschritten eine Pause eingelegt werden soll und wie viele Millisekunden die Pause dauern soll.

Startadressen
Schließlich läßt sich noch festlegen, ob der Abstand der beiden Programme zufällig gewählt werden soll (im Turnier-Modus unter Berücksichtigung des Mindest-Abstandes). Wenn nicht, beträgt der Abstand immer die halbe Spielfeldgröße.

5. Der Einzelschrittmodus

Mit der rechten Maustaste gelangen Sie in den Einzelschritt-Modus. Bei jedem Druck auf die rechte Maustaste wird ein Befehl abgearbeitet. Mit der linken Maustaste können sie den Inhalt einer Speicherstelle ansehen. Mit beiden Maustasten gleichzeitig gedrückt können Sie den Inhalt der Speicherstelle verändern.

8. Für Einsteiger: der Assistent

Vielleicht sind sie jetzt durch so viel Theorie abgeschreckt worden. Dann möchten wir Sie jetzt ein bißchen aufbauen. Damit das Entwickeln von CoreWar-Programmen einfacher geht, haben wir einen Assistenten programmiert. Im Assistenten müssen Sie nicht das Programm als “Text” wie bei klassischen Programmiersprachen schreiben, sondern setzen in einem Dialogfenster einen Assemblerbefehl (per Drop-Down-Liste) aus seinen Bestandteilen zusammen. Zu dem gerade ausgewählten Befehl werden immer die genaue Bedeutung und die benötigten Argumente angegeben. Ein Erlernen der Programmiersprache Redcode kann der Assistent allerdings auch nicht ersetzen.

9. Ausblicke

Was nützt es, wenn jeder alleine zu Hause ein paar gute CoreWar-Programme geschrieben hat? Interessant wird CoreWar erst dann, wenn die Kampf-Programme verschiedener Autoren gegeneinander antreten. Zur Realisation schweben uns dabei folgende Ideen vor:

* Am Anfang werden wir die Adressen von allen CoreWar-Spielern sammeln. Jeder kann sich auf die Liste setzen lassen, wenn er will, und sich (gegen 2 DM Rückporto oder per eMail) die Liste zuschicken lassen. So können die Spieler untereinander in Kontakt treten.
* Wenn es genügend Spieler gibt, wird es Turniere geben, bei denen alle CoreWar-Programme gegeneinander antreten.
* Vielleicht findet sich ja auch ein Mailbox-Bereiber, der ein CoreWar-Brett einrichtet oder sogar Online-Turniere veranstaltet. (An dieser Stelle ein Aufruf an alle Sysops, die daran Interesse haben: Meldet Euch)
* Wenn wir wieder etwas mehr Zeit haben (und genügend Registrierungen erhalten haben), werden wir eine Internet-Version von CoreWar herausbringen.

10. Die Programmierschnittstelle

Das Betriebssystem, unter dem die Redcode-Programme ablaufen, heißt MARS. MARS steht für “Memory Array Redcode Simulator”. Das gesamte MARS-Betriebssystem befindet sich in der Dynamic Link Library MARS.DLL. Wenn Sie nicht nur CoreWar-Programme schreiben wollen, sondern “echte” Windows-Programme, die CoreWar-Spieler bei der Arbeit unterstützen sollen (z.B. einen Debugger), können Sie auf unserem MARS-Betriebssystem aufbauen. Die Syntax der Befehle finden Sie in der Unit USEMARS.PAS. Natürlich brauchen Sie sich nicht auf Turbo-Pascal beschränken, sondern können in jeder anderen Programmiersprache ihre eigene CoreWar-Oberfläche programmieren. Als Beispiel zum Ansteurern der DLL finden Sie DEMOWAR.PAS ebenfalls in diesem Verzeichnis.

11. Autor

Copyright und Vertrieb:
Barbara Tikart Polarwolf Hard & Software
Am Stadtwald 45, 63906 Erlenbach am Main
Tel.: 09372 / 4198

Programmiert hat dieses Programm:
Andreas Tikart
Hechtgang 6, 78464 Konstanz
Tel.: 07531 / 33868
eMail: Andreas.Tikart@uni-konstanz.de
http://www.uni-konstanz.de/studis/asta/software

Dieses Programm wird als Shareware vertrieben. Sie dürfen CoreWar 30 Tage lang unverbindlich testen. Nach Ablauf dieser Frist müssen Sie sich für 30 DM registrieren lassen oder das Programm wieder von Ihrer Festplatte löschen. Das Programm darf kopiert und weitergegeben werden, solange nichts am Programm verändert wurde.

AnhangA: Eine Einführung ins Programmieren

Für alle, die bisher mit Programmieren noch nichts am Hut hatten, folgt eine kleine Einführung in die Programmiersprache Redcode und das Programmieren überhaupt.

1. Wie funktioniert eine Programmiersprache ?

Jedes Programm ist eine Ansammlung von Befehlen. Ein Befehl ist eine einzelne Anweisung an den Computer. In der Programmiersprache BASIC bedeutet der Befehl

PRINT „HALLO WELT“

daß der Text „HALLO WELT“ auf dem Bildschirm ausgegeben werden soll (wenn man des englischen mächtig ist und weiß, daß PRINT im Deutschen DRUCKEN bedeutet, liegt die Wirkung doch klar auf der Hand).
Der Befehl selbst heißt in diesem Fall „PRINT“; die Daten, die der Print-Befehl benötigt, heißen „Argumente“ und folgen direkt auf dem Befehl (durch ein Leerzeichen getrennt). Der Print-Befehl besitzt genau ein Argument, nämlich den Text „HALLO WELT“.

Ein kleines Programm in GW-Basic sieht dann z.B. so aus:
```
10 PRINT  „HALLO WELT“
20 GOTO 10
```

Das Programm wird zeilenweise abgearbeitet. Die Zahl vor dem Befehl läßt sich mit einer Zeilennummer vergleichen.
Zuerst wird der Befehl PRINT „HALLO WELT“ ausgeführt. Das Ergebnis ist uns ja schon bekannt. Sobald dieser Befehl fertig ausgeführt wurde, wird der nächste Befehl in Angriff genommen. Hier heißt dieser Befehl GOTO 10. Der Computer wird mit diesem Befehl angewiesen, bei der Zeile mit der Nummer 10 weiterzumachen. In dieser Zeile findet der Computer wieder den Befehl PRINT „HALLO WELT“ und es wird noch einmal den Text „HALLO WELT“ auf dem Bildschirm ausgeben. Die auf Zeile 10 folgende Zeile ist wieder Sprung-Befehl, der den Computer wieder zu Zeile 10 zurücklenkt. Der Computer wird also immer wieder den Text „HALLO WELT“ ausgeben, bis es dem Computer-Benutzer zu dumm wird, und er das Programm abbricht.

2. Das erste Redcode-Programm

Wie müßte das oben geschilderte Programm in Redcode aussehen? Bei dem GOTO-Befehl ist das ganz einfach. In Redcode heißt dieser Befehl JMP, wird aber fast genauso verwendet. Da Redcode aber keine Ausgabe auf dem Bildschirm vorsieht, wollen wir statt dessen etwas in den Speicher des Computers schreiben. Der Speicher eines Computers ist in Zellen aufgeteilt, jede Zelle kann genau einen Redcode Befehl (entspricht einer „Zeile“) aufnehmen. Das Programm sieht dann so aus:
```
MOV    #1    5
JMP    -1
```

Der erste Befehl ist die Abkürzung von „Move“ und bedeutet „bewege“. In diesem Fall wird die Zahl 1 in die Speicherstelle 5 geschrieben. Wo die Speicherstelle 5 liegt, werden wir später noch besprechen. Wichtig zu wissen ist: Der MOV-Befehl besitzt zwei Argumente: Ein Quell- und ein Zielargument. Wenn als Quelle eine Zahl verwendet werden soll, muß dieser Zahl ein Nummernzeichen ‘#’ vorangestellt werden. Wenn das Nummernzeichen fehlt, ist mit der Zahl nur die Adresse der Speicherstelle gemeint, in die die Zahl geschrieben werden soll, bzw. aus der die Zahl geholt werden soll.

Damit das Programm funktioniert, muß der JMP-Befehl lauten: Springe um eine Zeile zurück. Jetzt wird auch das -1 klar: Es werden nicht, wie bei BASIC, absolute Zeilennummern angegeben („Springe zu Zeile 10“), sondern nur relative Angaben („Springe um eine Speicherstelle zurück“). Bei dem MOV-Befehl geht die Speicheradressierung übrigens analog. Mit MOV #1 5 ist als Ziel die Speicherstelle gemeint, die vom MOV-Befehl 5 Speicherstellen in Vorwärtsrichtung entfernt ist.

3. Wie sag ich’s meinem Computer?

Als nächstes stellt sich die Frage, wie bringe ich dieses Programm meinem Computer bei? Starten Sie dazu den Redcode-Compiler. Der Redcode-Compiler funktioniert wie ein ganz normaler Texteditor. Sie können die beiden Programmzeilen wie einen ganz normalen Text eingeben. Wenn Sie damit fertig sind, können Sie mit „Compile“ das Programm compilieren. Compilieren bedeutet, daß der Text in ein vom Computer lesbares Format umgewandelt wird. Natürlich können Sie das Programm auch mit dem Assistenten eingeben.

Wenn das erledigt ist, starten Sie CoreWar. Wählen Sie als Programm ROT Ihr soeben geschriebenes Programm aus, und als „Gegner“ (BLAU) das Programm ANTIIMP, welches unserem Programm nicht in die Quere kommen wird (Schalter „Programm ROT festlegen“ bzw. „Programm BLAU festlegen“). Legen Sie noch mit „Optionen“ die Feldgröße auf 1024 fest.

Sie sehen jetzt im oberen Teil des Gitters zwei rote Kugeln, von denen sich die Linke mit einem dunkelroten Quadrat markiert ist. Dies ist Ihr Programm. Wenn sie mit der Maus auf eines dieser Felder gehen und die linke Maustaste drücken, wird der Inhalt dieser Speicherstelle angezeigt - probieren Sie es aus!

Mit der Rechten Maustaste wird immer ein Befehl abgearbeitet, abwechselnd von Ihrem Programm und dem blauen Gegner. Sie können jetzt richtig beobachten, wie Ihr Programm Schritt für Schritt abgearbeitet wird. Wenn es schneller gehen soll, müssen Sie den Schalter „Kampf Start/Stop“ drücken.

Als nächstes können Sie versuchen, das Programm IMP (s.o.) zu verstehen.