TR-H-268

人工生命システム Tierra

木目沢　司，トーマス・レイ

1999.3.4

ATR人間情報通信研究所

〒619-0288　京都府相楽郡精華町光台2-2　TEL: 0774-95-1011

ATR Human Information Processing Research Laboratories

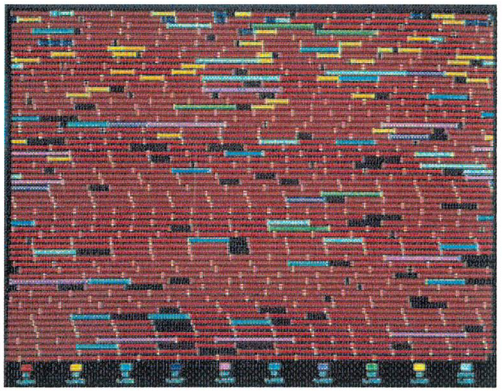
2-2, Hikaridai, Seika-cho, Soraku-gun, Kyoto 619-0288, Japan

Telephone: +81-774-95-1011

Fax : +81-774-95-1008

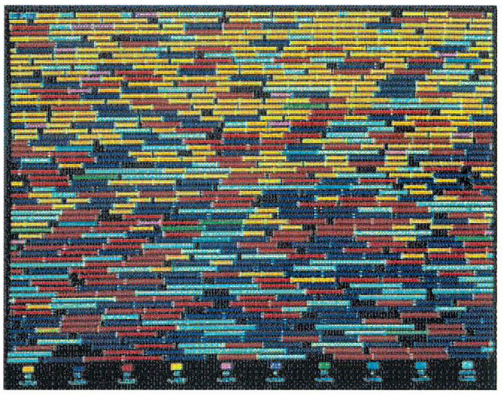
口絵1 デジタル生物の進化

Artificial Life Monitor(ALmond) programによるシミュレーション画面
(Marc Cygnus氏(mcygnus@mcs.com)作成)



1. まず、先祖種(赤色)がスープメモリを満たす。寄生種(黄色)はまだわずかである。

2. 寄生種はコピープロシッジャーを持たない。寄生種は先祖種よりコードが短いので自己複製効率が良く、次第に先祖種を圧倒し始める。

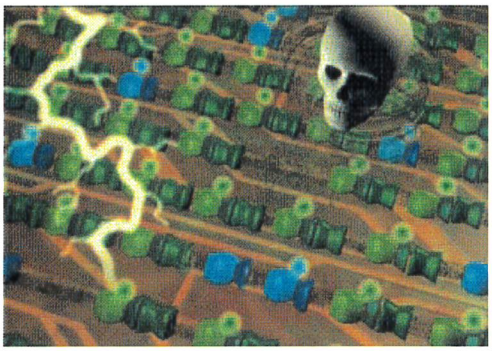


3. やがて先祖種の中から、寄生種の攻撃を逆手にとって自己増殖する種(ハイパー寄生種(青色))が現れる。

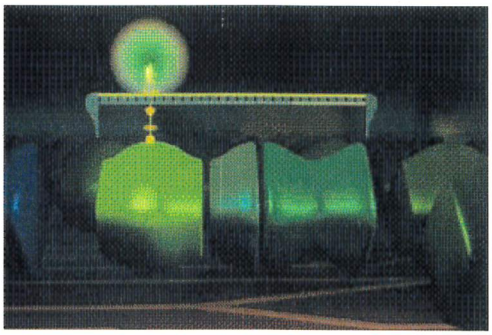
4. 寄生種が絶滅し始め、ハイパー寄生種がスープメモリの大部分を占めるようになる。

口絵2 デジタル生物の仕組み

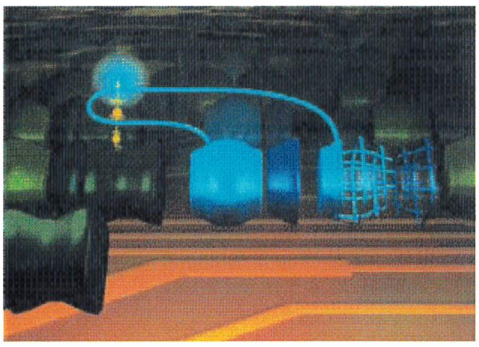
Image by Anti Gravity Workshop: Coutesy of the Santa Fe Institute



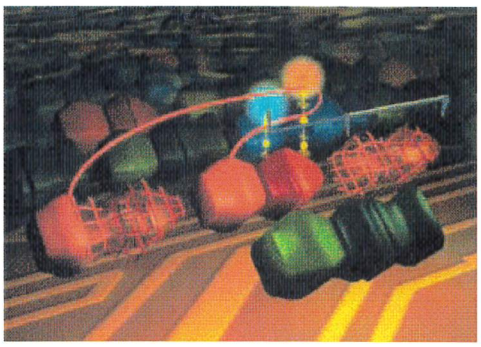
1. Tierraの実行環境のイメージ図
コンピュータ・メモリに緑色や青色の物体で描かれている自己複製生物(デジタル生物)達が存在している。
デジタル生物達は時折、雷を受けてその遺伝子(自己複製コード)が突然変異する。
またメモリがいっぱいになると死神(リーパー)が現れ、古くて欠陥の多いデジタル生物を殺す(淘汰)。



2. 先祖種のプログラム(遺伝子)
先祖種のプログラムは三つの部分に分かれる。CPU(緑の玉)は今、最初の部分のプログラムを実行している。これはこの生物のプログラムのサイズを測定しているところである。



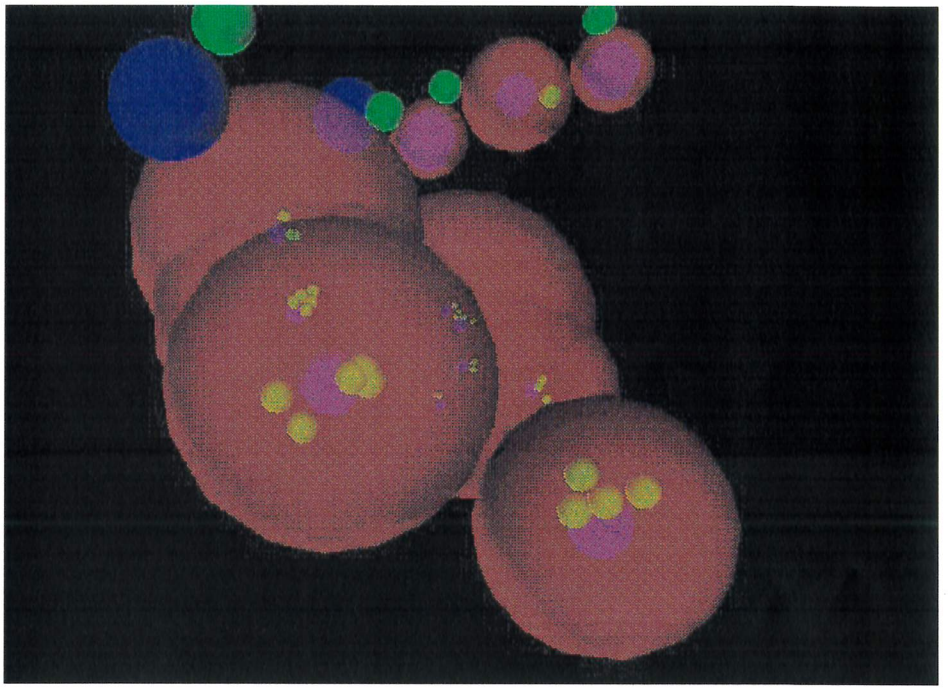
3. 寄生種(二つの青い物体)
寄生種は先祖種が持っている三つのプログラム・コードブロックのうち、コピー・プロシッジャーを持たない。そこで近くにいる先祖種のコードを利用して自己複製する。図はワイヤーフレームで描かれている娘生物の自己複製を行なっている途中の様子である。



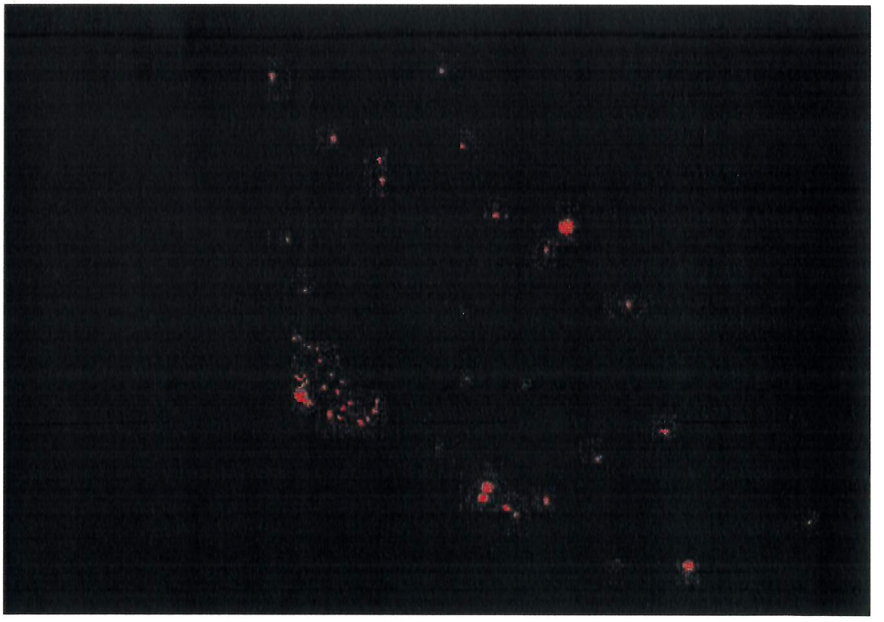
4. ハイパー寄生種(赤い物体)
ハイパー寄生種は寄生種のCPU(青の球体)を使って自己複製できる。自分自身のCPU(赤の球体)と合わせて同時に二つの娘生物を作ることができる。

口絵3 デジタル生物から見たサイバースペース

Tierraをインターネットで結んだNetwork Tierra Projectが進行中である。Network Tierraのデジタル生物はTping命令という特殊な命令語を使って、自分が住んでいるサイバースペースを観測することができる。以下の図はこのTping命令により得られたデータをVRML(Virtual Reality Modeling Language)により表示したものである。



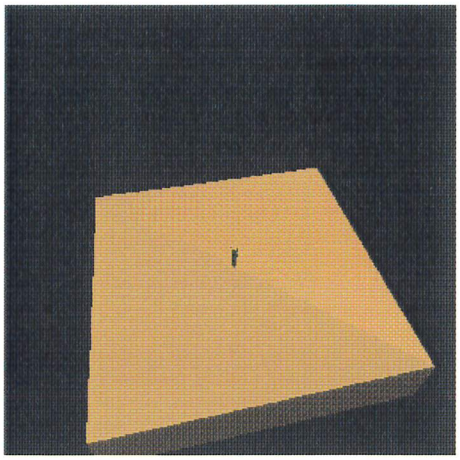
自分の住んでいる近くのTpingデータのVRML表示。青い球体は仮想ネットワーク上の各ノードのCPUの処理スピードを表している。赤い球体は各ノードのスープメモリのサイズを表している。緑の球体は各ノードに生息しているデジタル生物の数を表している。緑の球体1個について100匹の生物が生息していることを意味している。



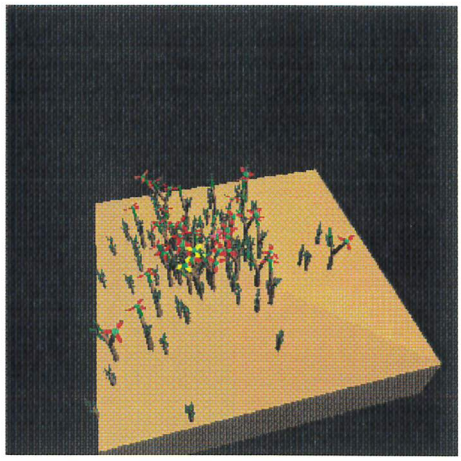
Network Tierra全体のTPingデータのVRML表示。

口絵4デジタル生物による仮想森の進化

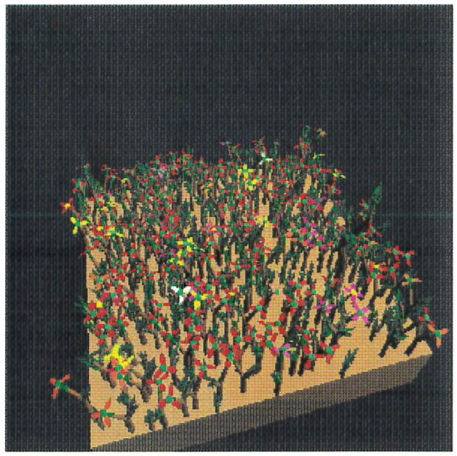
形状を生成するデジタル生物によってできた“仮想森”の進化シミュレーション。シミュレーションが進む毎に次第に複雑な樹木の形状が現れる。



0 Step

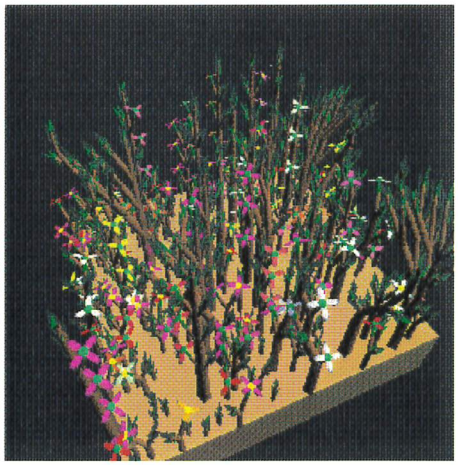


16 Step



28 Step

52 Step



2016 Step

12096 Step

もくじ

第 1 章

はじめに

本テクニカルレポートではトム・レイ博士が中心になって行われている人工生命研究プロジェクト「Tierra」（ティエラ）の研究成果とその技術内容を紹介する。

Tierraは人工生命研究の中では最も良く知られているが、実際にはどんな仕組みや原理でできているのかといった事は意外と理解されていない。これは、Tierraが生命や進化の本質とは何かというトム・レイ博士の深い思索から生まれたものであり、これを本当の意味で理解するには、我々の周りの生命現象に対する理解が必要だからである。具象的な生命現象からその本質的な部分のみを抽出し、コンピュータという道具を使って再構成することにより、生命と進化をより深く理解しようというのがTierraのアプローチである。

本レポートでは「Tierraって何？」と疑問を持たれた初心者の方から、これからTierraをヒントに何か研究してみようという上級者の方までの幅広い読者を対象にTierraのやさしい解説からソースコードにいたるまで、広範囲な説明を行った。

全体の構成は主に3つの章から成り立っている。第2章ではTierra解説と題して、Tierraの概念、しくみ、研究成果をできるだけ平易に解説する。この章の内容はWWWホームページ「Tierra入門」(http://www.hip.atr.co.jp/~kim/TIERRA/tierra.html)を元にしている。

第3章ではTierraをより深く知りたい人のための技術情報を提供する。この章はtierra.doc[Ray98b]の一部を邦訳したものである。Tierraの入手方法とイントール、起動方法そしてソースコードの解説を行う。実際に手元のパソコンでTierraを動かしてみたいという方はこの章をご覧いただきたい。

また、第4章では著者自身が行ったTierra応用システムについて紹介する。

第 2 章

Tierra 解説

2.1 Tierra のはじまり

1990年1月、米国デラウエア大学の**進化生物**学者であるトム・レイは、コンピュータ内部の世界に進化するシステムを実現した。ダーウィンが唱えた**進化論**はこれまで化石などの進化の産物を観察することで、その理論の検証と新たな理論の展開がなされてきた。トム・レイは1970年代末以来毎年中米のコスタリカの熱帯雨林で、進化の形跡を求め観察を続けていた。だが、彼は次第に不満を感じ始めていた。確かに熱帯雨林の中では動植物の複雑な相互関係や形態変化が見いだされ、それが進化の結果であろう事は観察できた。しかし、進化のプロセスそのものの進行を調べるためには観察者である人間の寿命はそれに比べてあまりにも短すぎる。

1987年、彼は東芝のラップトップパソコンを買った。彼はそれを使って、ハーバード大学の大学院生だったころに思いついた自己複製するプログラムを作ろうと考え始めた。やがて彼は「**人工生命**」[ALife]という本を知り、第1回人工生命国際会議を組織したクリス・ラングトンに出会う。約1年にわたりトム・レイとラングトンはE-メールにより意見の交換を続けた。レイの提案は**突然変異**と**遺伝コード**の組み替えにより、自己複製するプログラムをつくるというものだった。ラングトンはこの提案に対して、トム・レイのやろうとしている事はコンピュータウィルスと同様の危険性があることなどを指摘した。

1989年トム・レイは第1回人工生命国際会議が開催された米国ニューメキシコ州ロスアラモスを訪れた。そこには**複雑系**や人工生命の第一線の研究者が集まっていたが、彼の計画にはラングトン同様、懐疑的な意見が多かったという。

彼らの第一の懸念はその自己複製プログラムが、コンピュータの外に漏れ出す危険性があった点である。第二に既存のコンピュータ言語においてはコードの一部が変異しても元の機能を維持できるようなものは皆無で、そのような試みはかつて尽く失敗してきたという点である。

第一の懸念に対して、ラングトンは仮想コンピュータの上で自己複製プログラムが動くようにするという提案をした。普通のコンピュータではハードウェアの上にオペレーティング・システムがあり、その上で各種のプログラムを実行する。ラングトンの提案は、**オペレーティング・システム**の上に仮想的なコンピュータとしてふるまうプログラ

ムを実行し、自己複製プログラムはさらにその仮想コンピュータの中で実行させようというものである。こうすることにより、自己複製プログラムは仮想コンピュータの外へ逃げ出すことはなくなる。

第二の懸念に対しては、もともとコンピュータ技術者ではなかったトム・レイにとっては問題とは考えられなかった。実際自然界では遺伝子が変異し進化が起っている。遺伝子が情報を伝えるコードであるにすぎないとすれば、コンピュータ・プログラムが進化していけないはずはない。彼はそう考えた。デラウエア大学に戻ったトム・レイは彼のアイデアの実現に取りかかった。

既存のコンピュータ言語では、そのコードに一部でもランダムな改変を施せば、まったく機能しなくなる。しかし遺伝コードはロバスト（頑強）である。この違いはなぜなのか。トム・レイは二つのアイデアを実行した。

第一は“命令セットの数”である。遺伝コードは右図のようにU(ウラシル）、C（シトシン）、A（アデニン）、G（グアニン）の4種類の塩基の組合せで表現される。三つの塩基をひとつのグループ（コドンという）として、それがひとつのアミノ酸に翻訳される。三つの塩基で64通り(4の3乗)のコドンがあり、それぞれのコドンは20種類のアミノ酸（または終了命令）のどれかに翻訳される。このことは遺伝コードには冗長性があり、同じアミノ酸に翻訳されるコドンが複数あることを意味する。

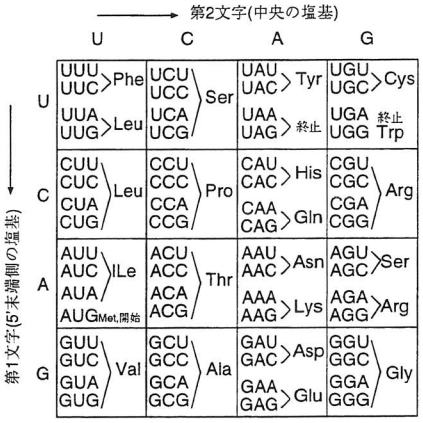


Figure 2.1: 遺伝暗号表

ここで重要なのは、突然変異は塩基の置換、挿入、削除を起し、それにより64種のコドンの置換が起こり、さらにはアミノ酸の置換とそれから作られるたんぱく質に変異を起すということである。

同様にコンピュータを動作させるプログラムを**機械語**として解析してみると、その突然変異は非常に広範囲の対象に対して起こりうることがわかる。たとえば、最近普及している多くのコンピュータで使用されている機械語の命令語は32ビットで構成されるが、このことは2の32乗(約40億以上)の命令語の置換がありうる事を意味する。40億個の命令セットの中からと、20個の命令セットの中からとでは、有用な命令セットを見つけられる可能性があまりにも異なる。

そこでトム・レイは命令語のサイズを5ビットに縮小した。そうすれば命令語の種類は2の5乗すなわち32個になる。こんな少ない数の命令語ではコンピュータを動作させるには不十分と思われるかもしれないが、そうではない。実際のコンピュータの機械語でよく使われるのは、せいぜい100種類くらいの命令語である。32ビットの命令語用のアドレス空間のほとんどは、数値や、その命令が作用される**オペランド**（主命令が対象とする相手、条件、パラメータ）を指定するのに使用される。例えば、**Table:2.1**のような8086用CPU（これは16ビットであるが）の機械語では**オペコード**（主命令）は100個程度で、オペランドを作用させる事により様々な計算処理をさせる。

ペコード	オペランド	説明
MOV	AL,1A	ALレジスタに値1AHを転送せよ
MOV	CA,0100	CXレジスタに値0100Hを転送せよ
MOV	D1,0200	D1レジスタに値0200Hを転送せよ
CALL	010D	アドレス010DHのサブルーチンを呼べ
INT	20	20番の内部割り込みをかけよ
CMP	AL,[D1]	ALレジスタとD1レジスタの指すアドレスのメモリの内容を比較せよ
JZ	0114	ゼロフラグが1ならばアドレス0114Hへジャンプせよ
INC	D1	レジスタD1を1増加せよ
LOOP	010D	CXの値を1減らし、0でなければアドレス010DHへジャンプせよ
RET		サブルーチンから復帰せよ

Table 2.1: 簡単な8086CPUアセンブラプログラムの例

トム・レイは機械語からオペランドを取り除いた。彼が考え出した機械語ではすべての数値はCPUのレジスタの中に格納されるか、レジスタによってその値のあるアドレスを指示する事で示されるようにした。

この変更は別の問題を引き起こした。しかし同時にまた新たな発明をもたらした。コンピュータはメモリ空間に並んだ命令を順番に行っていくが、ときどき分岐やループといった命令が行われる。つまり命令語の実行順序がメモリのある位置から別の位置にジャンプすることが頻繁にある。

コードの一部分が別のアドレスにある一部分と相互作用を起す。トム・レイはこのような事が生物学的にはどのように類推できるか考えた。生物の世界では具体的な座標空間を計算してたんぱく質分子の相互作用が行われているわけではない。そのかわり、たんぱく質分子は鍵と鍵穴の関係により分子の表面形状が適合するかどうかで相互作用がおこなわれている。表面の相補的な形状により分子同士が結び付く。トム・レイはこの考えを彼の機械語に取り入れた。これが2番目のアイデアである。正確なアドレスを指定するかわりにコードの特定部分にパターンを設け、そのパターンと最も近くにある相補的なパターンを探し、そこにジャンプするのである。

こうして、これら2つのアイデア(少数の命令セットと相補的なパターンによるアドレッシング)を盛り込んだ仮想コンピュータ“Tierra”(スペイン語で地球を意味する)が誕生した。そしてトム・レイはすでに“本当の”コンピュータの機械語用に作成していた自己複製プログラムをTierra用に移植した。

トム・レイは最初の段階としてこの仮想コンピュータTierraと自己複製プログラムが、自己複製以外に特別な事ができるようには設計していなかった。彼はその頃、高度な自己複製能力をもった仮想コンピュータを作るには何年もかかるだろうと思っていた。だが、1990年1月3日の夜、はじめて仮想コンピュータの上に自己複製プログラムの実行に成功したときその計画は根本的に変更された。トム・レイの仕事はその晩、システムの設計者から再び観察者に変わった。ただし今度は自然界の熱帯雨林のではなく、“デジタル・ジャングル”の観察者にである。

このデジタル・ジャングルの中では生物たちは近隣の生物たちを搾取したり、あるいはその攻撃を防御する方法の探索などが終わることなく繰り返されていた。進化は基本的には利己的なプロセスである。その成功度は遺伝子を将来の世代にどれだけ多く残せるかによって評価される。その目的を達成するのに進化は非常に独創的なもので

ある。

一旦環境が生物(トム・レイが自分でコーディングした種で**先祖種**と呼ぶ)で満たされると、これらの生物たち自身が環境の中でもっとも重要なものとなる。彼の仮想コンピュータが最初に実行された夜、生物たちは彼らの環境には多くの情報がある事を見つけ出した。あまり賢くない生物たちはそれらが必要としている情報すべてを単純に複写していた。しかしより進化した種が現れ、近隣にある情報を使って自己複製する方法を編み出した。そしてそれらはすぐにデジタルジャングルの環境を満たした。これらを先祖種のコードを利用して自己複製を行うことから**パラサイト**（寄生種）とよぶ。

だがこれらパラサイトの繁栄は長くは続かなかった。最初パラサイトは必要とする情報を簡単に見つける事ができた。しかしそれらが増えて環境全体を満たし、**ホスト**(先祖種)の数がわずかになると、その情報を見つける事は困難になる。この種は急速に数を減らすことになる。

パラサイトが減るとホストは再びその数を回復させる。するとまたパラサイトの数がその後を追って回復する。このようにパラサイトとホストによる**Lotka-Volterra**サイクルと呼ばれる捕食者と獲物の個体数の周期変動を起す現象が観察された。

だがこれだけではなかった。ホストはやがてパラサイトに対して“免疫”的な働きをするメカニズムを発達させた。まずホストは一度目はパラサイトの複製を許す。そしてパラサイトをだます情報を用意する。この情報はパラサイトが 2 回目の複製過程でそれらが寄生しているホストの複製を行わせるようになっている。ホスト自身は自己複製するときこのパラサイトはなくてもかまわないので、やがてこのパラサイトは絶滅する。この種をトム・レイはパラサイトが持っているエネルギー(CPU 時間)を奪ってしまう種という意味で**ハイパー・パラサイト**と呼んでいる。

パラサイトが絶滅した後、ハイパー・パラサイトはさらに進化してソーシャル・ハイパー・パラサイトになった。もはやパラサイトは存在しないので、それらはお互いに近隣に位置してお互い同士の増殖を助け合う。だがすぐに、ソーシャル・ハイパー・パラサイト同士の信頼関係はトム・レイが**チーター**(だまし屋)と呼ぶ種によって壊される。チーターはハイパー・パラサイトが使ったのと同様な方法で、仲間のエネルギーを奪ってしまう。

その後、さらなる実験で Tierra の仮想生物たちは sex（性）さえ発見した。デジタル生物同士で遺伝子を混ぜ合わせて、新たな遺伝コードをもった子孫を産み出した。

トム・レイは Tierra は「生命がどのような形をとりうるのだろうか?」、そして、「どうして種はそれぞれ異なっており、種と種の中間の形をもった生物がいないのはなぜなのか?」といったような疑問に答える強力な道具となるだろうと言う。

トム・レイの最大の夢はデジタル・コンピュータの中で**カンブリア爆発**を起すことである。6 億年前に起ったカンブリア爆発は原始生命の誕生以上に特筆すべき出来事である。このとき突如としてありとあらゆる多細胞生物が出現した。そして割と短い期間内で生物の色々な形が試され、その中から今日まで残っている主要な生物の形が決められた。Tierra においてもしカンブリア爆発と同様な現象を起すことができれば、それはあらゆる種類の多様性や複雑さをもった情報処理システムをもったデジタル生物の創造が行われることを意味するだろう。

2.2 Tierraのしくみ

我々人間は生命というものについて、たった一つの形態しか知らない。地球というひとつの生態系しか我々は知らないからだ。ここにもう一つの生態系を示す。この生態系はコンピュータのなかに作られている。有機化合物でできた生命とは全く別の、デジタル情報でできた生態系“Tierra（ティエラ）”である。

2.2.1 基本概念

有機生物（植物）は太陽エネルギーを利用してこれを化学エネルギーに変換し、栄養物を自分の体を構成するための物質に作り変えている。草食動物はもちろん、肉食動物が獲物を食して得るエネルギーも遡れば植物が得た太陽エネルギーに行きつく。地球上の生態系を極単純化すれば、あらゆる生物は限られた量のエネルギー(太陽エネルギー)と養分としての物質を獲得するために、それらが占める空間を奪い合っていると考えられる。

Tierraは地球生態系のこの基本的な仕組みをまねている。コンピュータの中の資源として最も重要なものが2つある。

一つは中央演算処理装置(**CPU**)の命令実行スピードで、速ければ速いほどたくさんの情報処理が短い時間で行える。これは実在の生物においてエネルギー（日光、食物を分解して得られるエネルギーなど）がたくさん得られれば得られるほどたくさんの化学反応を起して、自分の体を作ったり子孫を残したりなどの多くの生命活動ができることに対応する。

もう一つはメモリー空間(**RAM**)でコンピュータで実行されるプログラムやそのためのデータはRAM上に置かれる。大きく複雑なプログラムほど大きなメモリー空間を必要とする。これはちょうど体の大きな生物ほど、広い空間とその体を構成する物質を必要とするのと同じである。

Tierraの世界では生物(デジタル生物と呼ばれる)たちはそれぞれ、独立したプログラムとそれを実行するCPUである。それらのプログラム(デジタル生物の遺伝子)たちは基本的な機能として自己複製能力がなければならない。彼らの目的は自分と同じプログラム（遺伝子）をより多くメモリ空間上に増やし続けていくことである。デジタル生物たちは同時に複数個存在するが、一般のコンピュータシステムではCPUは一つしかないのが普通である。Tierraシステムは**スライサー**と呼ばれる機構によって、それぞれの生物に順番にCPUの利用時間を割り当てる。より多くCPU時間を得た生物は生存競争においてそれだけ優位に立てる。

逆に自己複製がうまくできず、無効な命令ばかり実行するデジタル生物は**リーパー**と呼ばれる機構により殺される。これにより、より効率良く自己複製するデジタル生物が生き残りやすくなる。

Tierraの世界の生物の遺伝子はプログラムだと述べたが、それらは専用の機械語で記述されている。機械語はコンピュータを直接操作できる言語である。トム・レイによれば機械語はビット、バイト、CPUレジスタや命令ポインタを直接操作できるので、核酸というよりは、化学的に活性であることが可能なアミノ酸のようなものだという。

Tierraでは生物は**RNA**ワールドの生物たちのアナロジーで説明できる。RNAワールドの生物たちは遺伝情報を運び、代謝活動を行うからだ。

RAMメモリの中でデジタル生物たちが生息する部分をTierraではスープメモリと呼ぶ。デジタル生物たちのゲノムはひとつづきの機械語で構成される。それらは自己複製するためのコードを含む。トム・レイは最初の生物として80個の機械語で構成された先祖種のコードを書いた。

突然変異はある一定の割合でスープに置かれていた機械語コードを別の機械語コードに置換したり、デジタル生物が機械語命令を間違って行ってしまう事により起こる。

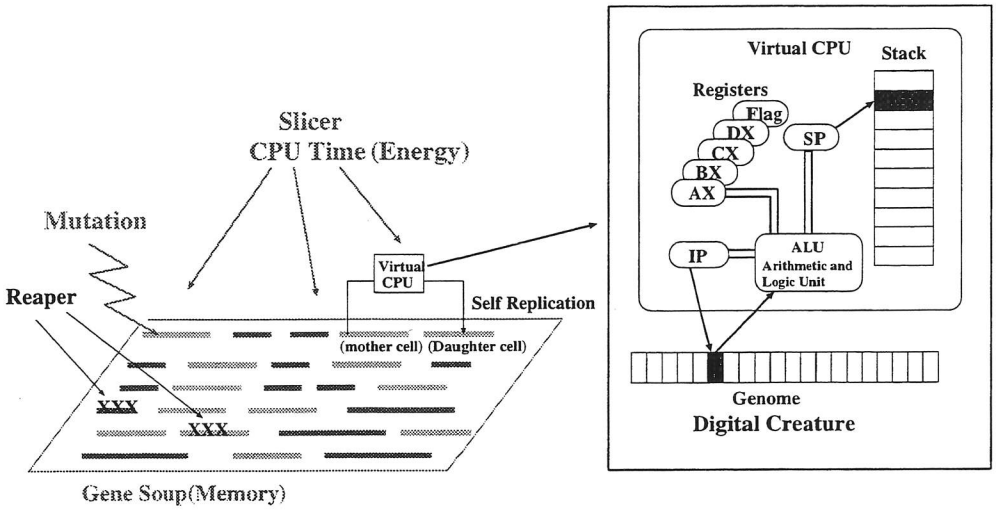


Figure 2.2: Tierraのシステム図

2.2.2 仮想コンピュータ

Tierraは仮想コンピュータである。本物のコンピュータから見るとそれは、**オペレーティングシステム**の上で実行されるプログラムの一つに過ぎない。しかしTierraプログラム自身は、一つの完全なコンピュータシステムの機能を有する。そしてデジタル生物たちのそれぞれの**プロセス**は、この仮想コンピュータ上の仮想プロセスとして実現されている。

Tierraを仮想コンピュータとして実現したのは、以下のようないくつかの利点があるからである。

1. もしデジタル生物が本物の機械語で記述され、それが本物のコンピュータの上で直接実行される事になれば、それはコンピュータウィルスや**ワーム**のように、コンピュータシステムに対して悪い影響を与える恐れがある。しかしデジタル生物が仮想コンピュータ用の機械語で記述されているならば、それは本物のコンピュータにとってはワープロソフトの文書ファイルのような単なるデータにすぎないのでコンピュータ自体にとっては安全である。

2. 機械語はそれぞれのハードウエア固有の命令セットにより構成されている。デジ

タル生物をある本物の機械語で記述してしまうと、そのハードウェアが技術の進歩により陳腐化した場合、もう一度新しいハードウエア用の機械語に書き直さなければならなくなるだろう。Tierraのプログラムは仮想コンピュータであるから、Tierra自体の移植さえできれば、デジタル生物の機械語(Tierra言語)を修正する必要はない。Tierra自体は**C言語**で記述されているので、ほとんどのコンピュータシステムに移植可能である。

3. 既存の多くの機械語は、当然ではあるがそれで記述されたコードが進化できるようには設計されていない。**フォン・ノイマン型コンピュータ**の機械語は、そのわずか一部でもランダムな変更を受けるとプログラムとして機能しなくる。したがって、コードに突然変異などの処理を施す事はほとんど不可能である。Tierraの仮想コンピュータで使用される機械語ではこのような“脆弱性”を回避するための工夫が凝らされている。

Tierraは**MIMD型並列コンピュータシステム**を意識して作られた。それぞれのデジタル生物が一つのCPUを持つよう設計されているが、これは実際はCPU時間を小さなタイムスライスに分割することにより、不完全ではあるが実現している。

この仮想コンピュータでは各CPUは2つのアドレス用レジスタ(AX,BX)、2つの数値用レジスタ(CX,DX)、エラーフラグ用レジスタ(Flag)、スタックポインタ(SP)、10ワード分のスタック、そして命令ポインタ(IP)を備えている(**Figure2.2**:Tierraのシステム図参照)。

ただしTierraの仮想CPUで実行される計算は突然変異を起こす方法の一つとして、極めて低い確率で無効にされる事がある。(**2.2.8**節: 突然変異参照。)

命令セットはCPU内のレジスタを使って、単純な数値演算やビットオペレーションを行うよう設計された。ある種の命令は、レジスタ内のデータをほかのレジスタに移動させたり、RAMメモリとレジスタ間でのデータの移動を行う。他に命令ポインター(IP)の位置を制御する命令も用意した。IPはデジタル生物が行っているコードのメモリ・アドレスの位置を特定するものである。

CPUは次々と、「**命令の読み込み**」→「**復号**」→「**実行**」→「**IPのインクリメント**」という一連のサイクルを繰り返す。IPによって指し示されている機械語命令がCPUに読み込まれ、そのビット・パターンを解読(復号)して対応する命令を特定し、その命令を実行する。そしてIPの値は1つ先の命令のアドレス値にインクリメントされる。次のサイクルでその命令がCPUに読み込まれ、復号、実行が行われる。しかし例えばJMP,CALL,RETのような分岐命令ではIPは離れた位置にある命令を指定する。

2.2.3 Tierra言語

進化可能な仮想機械言語(Tierra言語)は、生物のDNA,RNAとたんぱく質を参考にした2つの特徴をもっている。

一つは、命令セットの数を生物の遺伝コードの数と同じくらいのサイズになるようにした事である。実際の生物では64個のコドンを用いてDNAにコードされており、それ

らのコドンは20種類のアミノ酸に翻訳される。Tierra言語では5ビットの2進数で表現される32個の命令から構成されている。

また命令セットの数を本当の意味で減らすために、Tierra言語には数値オペランドが存在しない。Tierra言語ではレジスタとスタックだけを用いて数値演算を行う。何らかの理由で整数を命令コードの一部として使用したいときは、数値をレジスタに置くことにする。そしてその数値の修飾には最下位ビットの反転と左シフトを用いて行う。この事によってプログラムのサイズは増えるが、数値オペランドの必要はなくなる。

第二の特徴は、**テンプレートによるアドレス方式**である。世の中のコンピュータのほとんどの機械語命令ではデータの参照やジャンプ命令にはアドレス番号を用いる。一方生物の場合、例えば細胞質の中でたんぱく質分子Aが他のたんぱく質分子Bと相互作用しようとするとき、分子Bの正確な座標位置を必要とはしない。分子Aは自分の表面に分子Bの表面と相補的（凹凸がはまりあえるよう）な形状の「テンプレート」を用意する。そして拡散により2つの分子が近づいたとき、2つのテンプレートが相補的であればそれらは重なり合い相互作用を行う。

Tierra言語のテンプレートによるアドレス方式の例をjmp(ジャンプ)命令で示す。各jmp命令の後ろにはいくつかのnop(no-operation)命令が続く。nop命令にはnop0,nop1の2種類がある。例えば5つの命令からなるコード列、

```
jmp, nop0, nop0, nop0, nop1
```

を考えてみよう。このときシステムはこのコード列の前後にある最も近い位置にある相補的なパターン

```
nop1, nop1, nop1, nop0
```

を探す。もしパターンがみつかるとIPはそのすぐ次にある命令を指して実行を始める。もしパターンが見つからないときは、エラー・フラグをセットしてjmp命令を無視する。実際にはTierra実行開始時に探索範囲を指定しておく。

以上のようにTierra言語は2つのユニークな特徴を持っている。1つは数値オペランドを持たないごく少数の命令セットであり、もう一つはテンプレートによるアドレス方式である。その他の点では、ほとんどの典型的な機械語と同じようにできている。つまり、mov,call,ret,pop,pushなど普通の機械語が用意されている。この章の後節**2.7.1命令セット0の一覧表**に、Tierraで使用されている命令語の集まり（命令セット0）を示す。なお、命令セットは後にいくつか増やされた。初期のTierraで用いられた命令セットはこの命令セット0とほぼ同じである。

2.2.4 Tierraオペレーティングシステム

Tierraの仮想オペレーティングシステムはデジタル生物の棲息のために、(デジタル生物の)メモリとCPUタイムの割り当てなどを行う。

2.2.5 メモリの割り当て — 細胞性

Tierraはその実行開始直後、本物のコンピュータシステムのメモリのうち、その一部をデジタル生物が生息するための**スープ**用メモリーとして確保する。普通その大きさは60000バイトである(パラメータ設定ファイルにより変更可能である)。これはそのスープの中に60000個分の仮想機械語命令が置かれる容量がある事を意味する。(Tierra言語では各命令語は1バイトすなわち8ビットのうち5ビットしか使用しないが、プログラムでは1命令を1バイトに割り当てている。)

Tierraではデジタル生物の基本単位を「**セル**(細胞)」としている。各セルは1個または複数のCPUを持ち、またスープ内のあるメモリ・ブロックを確保して、そこに各セルのゲノム(Tierra言語のコードの並び)を保存している。このブロックへはそれを所有しているセルは書き込みおよび読み込み動作を行う事ができる。しかしそのブロックの所有者でないセルは読み込みだけしか行う事ができない。これは各セルが半透膜性の**細胞膜**で守られている事を意味する。

セルは自己複製時にはそのゲノムが記述されているブロックの他に、複製先セル(娘セル)用のブロックを確保する。これは仮想機械語命令"mal"によって行われる。そしてそのブロックに自分(母セル)のゲノムをコピーする。そして仮想機械語命令"divide"によって娘セルは独立(自分自身の命令ポインターを獲得)し、その時点で母セルはその娘セルのブロックへの書き込み権を失う。その後、娘セルも新たな母セルとなって自己複製プロセスを開始する。

2.2.6 エネルギー(CPU時間)の配分

自然界のように個々の生物たちが同時に活動するためには、Tierraオペレーティングシステムは**マルチタスク**(または並列処理型)でなければならない。Tierraではこの事を実現するために、CPU時間を分割(**タイム・スライス**)して順番にデジタル生物たちに配分する事で、この仕組みを提供している。Tierraシステムは**スライサー・キュー**と呼ばれる待行列を管理している。新たな生物(セル)が生まれると、仮想CPUが作られ、母セルのすぐ前のところの位置のスライサー・キューに入れられる。スライス・サイズがセルの複製に要する時間より十分に短ければ、擬似的に並列性が実現されていると考えてよい。

スライス・サイズの決め方であるが、それは各セルのゲノムの大きさの冪乗に比例するようになっている。もし冪を1より大きくすれば、ゲノムが大きい(遺伝コードが長い)生物は小さい生物より1命令語あたり、より多くのCPU時間が割り当てられることになる。もし冪を1より小さく設定しておけば、反対にゲノムが小さい生物ほど1命令語あたりより多くCPU時間が配分される。冪が1ならばゲノムのサイズには関係なくすべての命令語に等しくCPU時間が配分される。このように冪の設定の仕方で、大きな生物を好む環境にするか小さな生物を好む環境にするかを変える事ができる。一定の時間幅で時分割すると、大きな生物ほど各命令語に配分されるCPU時間は少なくなるので、小さな生物にとって有利な環境になる。

2.2.7 死と淘汰（リーパー）

自己複製する生物たちは、またたくうちにスープを一杯にしてしまう。もしこれらの生物たちが不老不死ならば、生物たちはあらたな複製を行う空間を得ることができなくなり、Tierraのプロセスは停止してしまう。

Tierraオペレーティングシステムは**リーパー**（死神）というシステムを用意している。これはスープがある程度一杯になると(普通80%以上に設定する。)、生物たちを殺すように仕掛けられている。

デジタル生物(セル)はそれぞれ直線状の待行列(**リーパー・キュー**という)にならべられている。あらたなデジタル生物が生まれると、それはリーパー・キューの最後尾に置かれる。リーパーが発動すると、それはリーパー・キューの一番上のデジタル生物から順に殺していく。しかしながら各生物は、それらが実行する命令語が失敗するか成功するかによって、リーパー・キューの中を上下するので、成功すればキューの中の位置が下げられて、死神の魔の手を逃れられる。失敗すれば上げられて、死神のえじきとなる時が近づく。

リーパーの導入によって、すぐれたアルゴリズムをもったデジタル生物はより長く生存し、劣ったデジタル生物の寿命は短くなる。

2.2.8 突然変異

進化はその生物の遺伝コード(**ゲノム**)が変化する事で起る。この変化(突然変異)は個々の生物が生きている間、もしくは次世代へゲノムを受け渡す途中の両方で起っているだろう。Tierraオペレーティングシステムでは、スープに置かれた命令語にランダムなビット反転を行ったり、Tierra言語を不完全なかたちで実行させる事によって突然変異を起している。

突然変異には二つの仕方がある。一つの仕方は、あるバックグラウンド・レートでスープ全体(例えば60,000命令語すなわち、300,000ビット)の中からランダムにあるビットを選びだし、そのビットを反転(0を1に、1を0に)させる。これは宇宙線によって地球上の生物の遺伝子が傷つけられることをまねている。この事によりいかなるデジタル生物も各命令が永遠に正常に実行されるという保証はなくなり、生物は不老不死ではいられなくなる。通常このバックグラウンド・レートは各世代毎に32個体中1個体が変異するように設定されている(パラメータ設定ファイルにより変更可能である)。

もう一つの突然変異の仕方は、デジタル生物が自己複製する最中に起る。複製するとき、ある確率で複製される命令語のビットを反転させる。これにより、複製エラーが起きる。その確率は通常、各世代毎に32個体中数個体が変異するように設定されている。どちらの場合も、突然変異が起きる間隔は周期的な現象が起るのをさけるために、ランダムに変化する。

これらの突然変異に加えて、Tierra言語の命令が低い確率で不正な動作をする事による変異が起る。ほとんどの命令セットに対し、命令の実行結果にある低い確率で1を加えるか減ずるかの操作を施す。例えば、加算命令はレジスタの値に1を加えるが、ときどき0または2を加えることがあるようにしておく。ビット反転命令は普通、最下位

ビットを反転させるが、ときどき最下位ビットの次の上位ビットを反転させたり、あるいはまったく反転をしない事があるようにしておく。また、左シフト命令は普通すべてのビットを1ビット左シフトさせるが、ときどき2ビットシフトさせたり、あるいはまったくシフト操作を行わない事があるようにしておく。このようにTierra言語の命令セットの動作はある程度の不確定さが入るようになっている。

突然変異の結果、新たな遺伝子型（ゲノタイプ）をもったデジタル生物が誕生する。それらはジーンバンク・マネージャーというシステムに自動的に登録される。遺伝子バンクは新たなゲノタイプをもったセルのうち、2回の自己複製に成功したセルに名前をつけて登録する。(これは1回目の自己複製プロセスと2回目以降の自己複製プロセスでは、命令実行回数が異なるためである。トム・レイが設計した先祖種の生物では自己サイズを測定するためのコードが含まれている。1回目の複製時には、スープメモリに複製領域を確保するために、自己サイズを測定しなければならない。その生物が2回目の自己複製を行うときにはサイズの測定は必要ないので、1回目と2回目では命令実行回数が異なる。)

それぞれのゲノタイプは、数字と3桁のアルファベットの組合わせで命名される。例えば80個の命令語でできた最初のデジタル生物には0080aaaという名前がつけられる。最初の数字が命令語の数を表し、次の3桁のアルファベットによりそのサイズで最初に現れた順番にaaa,aabというように命名する。最初に出現したサイズ45の生物は45aaaであり、次にもしサイズは同じ45だが遺伝コードの並び方が異なる生物が出現するとそれには45aabという名前がつけられる。

このようにそれぞれの生物に名前をつけ、同時にその生物の母生物の名前を関連つけておくことによって、進化系統樹を作成する事も可能である。遺伝子バンクは命名と同時にその生物が生まれた日時、そして“代謝”データ、すなわち、1回目と2回目の娘セルを自己複製するのに要した命令実行回数とエラー回数、そしていくつかの環境変数(テンプレートの探索限界パラメータおよびタイム・スライスの冪乗パラメータ。これらはその環境に適応する生物のサイズに影響する。)も記録する。

2.2.9 先祖種 (Ancestor)

ここでトム・レイが設計した最初のデジタル生物(先祖種0080aaa)のコードを調べてみよう。トム・レイはこの先祖種を設計するに際して、自己複製のためのコーディングのみを考え、意図的に進化を起させるような仕掛けは入れていないと述べている。

先祖種のコードは大まかにわけると三つの処理でなりたっている。なお、別紙付録「デジタル生物たちのコード・ブロック図」にこの先祖種や後で説明する進化の結果現れたいろいろな種のブロック図を掲載しているので必要に応じて参照していただきたい。

1. **自己検査**(サイズ計測)

   先祖種はまず、スープ内で自分のコードが開始される位置(コード開始アドレス)と終了する位置(コード終了アドレス)を計測する。コード開始アドレスは4つの

no-opration命令によるテンプレート(nop1,nop1,nop1,nop1)でマークされている。同様にコード終了位置はテンプレート(nop1,nop1,nop1,nop0)でマークされている。

先祖種は5つの命令adrb,nop0,nop0,nop0,nop0によってコード開始位置を特定する。adrb命令は4つのnop0命令と相補的なテンプレートを後方(スープアドレスが減少する方向)に向かって探索する。

同様にadrf,nop0,nop0,nop0,nop1により、コード終了位置を特定する。コード開始位置と終了位置はそれぞれBXレジスタとAXレジスタに格納され、AXレジスタの値からBXレジスタの値を減ずる事によって自分のコードサイズ(命令語の数)を計算し、CXレジスタに格納する。

2. 自己複製ループ

自己検査が終わると、自己複製ループに入る。まず、娘セルにコードを複製するためにスープ内に母セルのコードサイズ分の領域を確保する(mal命令)。このとき確保した領域の先頭アドレスをAXレジスタに格納する。そしてコピープロシッジャーを呼び出す。コピープロシッジャーを呼び出すには5つの命令語call,nop0,nop0,nop1,nop1によって、テンプレート(nop0,nop0,nop1,nop1)と相補的なテンプレートを探索する。これにより、コピープロシッジャーのコードにIP（命令ポインター）が移動してその処理を行う。

3. コピー・プロシッジャー

movii命令でコピー元のアドレス(BX)からコピー先アドレス(AX)にコピーする。この操作を母セルのコードサイズ(自己検査時にCXにセットされている)分繰り返す。

すべての命令語をコピーし終わると再び自己複製ループ内にもどり、divide命令を行う。この命令により母セルは娘セルのメモリへの書き込み権を失う。そして娘セルは命令ポインターを与えられる。そしてスライサー・キューとリーパー・キューにエントリーされる。

最初の複製が終わると、母セルは2回目以降は自己検査コードを実行しない。jmp命令により自己複製ループの始めに戻って再び複製を行う。このループは無限ループである。

この先祖種のコードを**2.7.2項**に掲載している。

その表では例としてコードの開始アドレスがスープの30000番地目に置かれているとして、命令ポインターの値(next ip)と、AX,BX,CX,DXの各レジスタにどのように値が格納されるかを紙上シミュレーションした。

この80命令のコードのうち48命令はno-operation命令である。4つのno-operation命令を組み合わせて12個のテンプレートをつくり、コードの開始、終了アドレス、コピー・プロシッジャーや無限ループ、ジャンプ命令のマーキングなどに使用している。

2.3 進化

さてここでTierraで起った進化の例をいくつか紹介する。ここで説明する進化の例は、サイズ60000(Tierra言語命令60000個分)のスープメモリに、前章で説明した80命令で構成された先祖種(0080aaa)を培養させた結果である。Tierraでは時間はTierra言語命令の全実行回数でカウントされる。先祖種は839命令で最初の自己複製を行い、2回目以降は813命令で娘セルを複製する。リーパーはスープメモリーの80%以上が命令語で埋め尽くされた後に起動される。一旦リーパーが起動されると、その後スープメモリーの占有率はおよそ80%に保たれる。

2.3.1 ビット反転による進化

ビット反転による突然変異の影響について考察してみよう。ビット反転による突然変異ではゲノムのサイズは変わらない。しかしビット反転によりある命令語が別の命令語になれば、自己複製プロセスに影響を与える。例えば、no-operation命令(nop0,nop1)にビット反転が起ると、以下のように起りうる5つの変異のうち1つは他のno-operation命令に、他の4つはまた別の種類の命令語に変わる。

```
命令語(16進, 2進表示)
nop0  (0x00, 00000)     -> nop1  (0x01, 00001)
                        -> not0  (0x02, 00010)
                        -> zero  (0x04, 00100)
                        -> incA  (0x08, 01000)
                        -> popA  (0x10, 10000)

nop1  (0x01, 00001)     -> nop0 (0x00, 00000)
                        -> shl  (0x03, 00011)
                        -> ifz  (0x05, 00101)
                        -> incB (0x09, 01001)
                        -> popB (0x11, 10001)
```

つまり、no-operation命令にビット反転が起ると80%の確率でテンプレートのサイズを変えてしまうか、または20%の確率でテンプレートのパターンを変えてしまう。テンプレートのパターンの変化はサイズ計測の誤り、複製ルーチンの呼び出しやループ、ジャンプ命令時に間違った命令ポインターを指してしまうなどの現象を起す。

Tierraでの典型的な進化のプロセスは通常、口絵1のようになる。なお口絵1はMarc Cygnus氏(mcygnus@mcs.com)が作成したArtificial Life Monitor(ALmond) programによるシミュレーション画面で、スープメモリの状態を示している。デジタル生物はこのスープメモリの中で、色のついた線分で表されている。線分の長さがデジタル生物のゲノムサイズ(コードサイズ)を示す。

この図のような進化のあと、やがて社会関係を作る（自己増殖するときにお互いに助け合う）種（社会的・ハイパー・パラサイト）や、その社会の間に割り込んで、ちゃっかり自分だけ増殖しようとするだまし屋（チーターと呼ばれる種）が現れる事もある。

なお、上記のような進化のプロセスはTierraの実行毎に必ずこうなると決まっているわけではなく、あくまで典型的な場合である。また、環境パラメータ（タイムスライス

の与え方や突然変異率）の設定の仕方や乱数のシードによっても、進化プロセスの傾向は異なってくる。

2.3.2 パラサイト（寄生種）

先祖種 0080aaa の 43 番目の命令語 nop0 にビット反転が起こり、テンプレートのパターンが変わると、興味深い新たな種が誕生した。43 番目の命令語は他の 3 つの no-operation 命令とともにコピープロシッジャーの始まりを示すテンプレートを構成している。このテンプレート (nop1,nop1,nop0,nop0) が (nop1,nop1,nop1,nop0) に変異してしまった。先祖種ではテンプレート (nop1,nop1,nop1,nop0) はゲノムの終了位置をを表す。したがって変異種は 43 番目の命令語が含まれるテンプレートをゲノムの終了位置と勘違いしてしまい、45 番目の命令語までしか複製しない。こうしてサイズ 45 の新種 0045aaa が生まれる。

この新種 0045aaa はコピー・プロシッジャーを持たない。したがって単独では自己複製を行うことができない。しかし Tierra の生物たちはスープメモリの半透膜的な性質により、他の生物のスープメモリ領域にある命令語を “読む” 事はできる。もし、0045aaa の近くに先祖種 0080aaa が存在しており、テンプレートの探索範囲内 (普通は 200 から 400 命令語分にセットされる) にコピープロシッジャーがあれば、0045aaa はそれを利用して自己複製を実行する事ができる。このように 0045aaa は先祖種 0080aaa の近くにいて、そのコードの一部を利用する事から、トム・レイはこの種をパラサイト (寄生種) と名付けた。

2.7.3項にパラサイト 0045aaa のコードを示す。

また別紙付録「デジタル生物たちのブロック図」の PARASITE の部分も参照されたい。

2.3.3 パラサイトに対する免疫

サイズ 79 のある種は、パラサイトに対する免疫機能を獲得した。0079aab というゲノタイプを持つ種の両脇にパラサイト 0045aaa を置いてそれぞれを繁殖させると、0045aaa は最初のうちは複製を行えるが、その後スープメモリーから排除されてしまった。0079aab がスープメモリーを支配した後、0051aao という新たなパラサイトが出現した。この種は何らかの方法で 0079aab の免疫機能を回避していると思われる。

トム・レイによると、この 0079aab と 0051aao いう遺伝子コードは彼のジーンバンクからは失われてしまったらしく、遺伝子コードは手に入らなかった。ただし、Tierra においてパラサイトに対する防御は割と容易に実現できる。たとえば、パラサイトはホストのコピー・プロシッジャーを call 命令により呼び出すのであるから、ホストのコピープロシッジャー・テンプレートが、パラサイトの持っていないテンプレートに置き換えられれば、もはやパラサイトはそのホストのコピー・プロシッジャーを呼び出す事はできなくなる。

2.3.4 ハイパー・パラサイト（寄生種への寄生）

0080gaiという種は、パラサイトの攻撃を逆に利用して排除してしまう事からハイパー・パラサイトと呼ばれる。0080gaiはコピー・プロシッジャーが終わってもret命令がないので、呼び出し元にリターンしない。そのかわりjmpb命令で自己複製ループの適切な場所に戻ってしまう。このようにして、この種はパラサイトのIP(命令ポインター)を奪ってしまう。さらに自己複製処理が終わる毎に、自己検査コードに戻ってコピー元のアドレスを格納しているBXレジスタと、コピーサイズを格納しているAXレジスタの値をリセットしてしまう。このようにしてパラサイトのCPUにはハイパー・パラサイトのコードをコピーするためのアドレス値とサイズが格納され、パラサイトはハイパー・パラサイトのコードをコピーしなければならなくなってしまう。

ここで説明した0080gaiのコードも失われてしまったらしい。そこで、ここではもう一つのハイパー・パラサイト0078aaaのコードを**2.7.4項**に示す。また、別紙付録「デジタル生物たちのブロック図」のHYPER-PARASITEにブロック図を示す。

パラサイトが0078aaaのコピープロシッジャーを使って1回目の全ゲノムのコピーを行った後、30170行目のjmpo命令で30147行目のdivide命令が実行され、1回目の自己複製には成功する。しかし30148行目のjmpb命令により、IPが30128行目のadrbにセットされてしまう。したがって2回目からは、パラサイトはこのハイパー・パラサイトの複製を行う事になってしまう。

さてこの種にはもう一つ注目すべき特徴がある。それは自己検査のコードである。先祖種ではテンプレートサイズをシフト命令などによって計算していたが、この種はadrb命令を使ってその手間を省いている。コードを見てわかるように最初はadrb,nop0,nop0,nop1により開始アドレスのテンプレートを求めている。これにより、AXレジスタの値は開始アドレス＋テンプレートサイズになる。このときadrb命令は同時にテンプレートサイズをCXレジスタに格納する。次にsubAAC命令によって、AX = AX - CXを計算し開始アドレスが計算される。次にmovBA命令によりBXレジスタにも開始アドレスをコピーする。そしてadrf,nop0,nop0,nop0命令によってコード終了アドレスを求め、結果をAXレジスタに格納する。次のjmpb命令は無視され、zero命令でCXレジスタを0にして、subCAB命令でCX = AX - BXによりサイズを計算し結果をCXレジスタに格納する。

2.3.5 社会的ハイパー・パラサイト

社会的ハイパー・パラサイト0061acgは集団になる事により、お互いに助け合いながら自己複製を行う。**2.7.5項**にコードを、別紙付録「デジタル生物たちのブロック図」のSOCIAL HYPER-PARASITEにブロック図を示す。

ハイパー・パラサイトは自己複製が終るとjmpb,nop0,nop0,nop1命令によって自分自身のサイズ計測コードにジャンプするが、このゲノムの場合はjmpb,nop0,nop1,nop0により自分の先頭コードを通り越して、自分の手前に存在する別の0061acgの末尾にあるjmpb,nop0,nop1,nop0,nop1のテンプレートを目印にしてジャンプする。Tierraではnop0,nop1,nop0の相補的テンプレートとしてnop0,nop1, nop0,nop1もパターンマッチング(最初のnop0

を無視してnop1,nop0,nop1と認識)している。こうして、このハイパー・パラサイトは再度、自己検査コードを実行できる。このゲノムがいくつか一続きになって並んでいれば、この手順が次々と繰り返されるのでスムーズな自己複製が可能になる。しかしもしもこのゲノムの手前に別のコードをもった種がいると、この手順は正しく行われない。普通のハイパー・パラサイトに対してこの種の有利な点はゲノム・コードが短くなる事である。とくにほとんどのテンプレートの長さが先祖種の4命令により短い3命令で済むように進化している。

2.3.6 チーター(騙し屋)

ゲノム0027aabはチーター(騙し屋)と呼ばれ、社会的ハイパー・パラサイトの間に割り込んで、そのIP(命令ポインタ)を自分のコードを複製させるようにする。またコピー・プロシッジャーも社会的ハイパー・パラサイトのコードを使う。

2.7.6項にチーター0027aabのコードを、別紙付録「デジタル生物たちのブロック図」のCHEATERにブロック図を示す。

2.3.7 高度な進化

これまで述べてきたデジタル生物たちの進化は、シミュレーション開始からおよそ10億命令くらいの期間で起った出来事である。ここでは150億命令後にスープメモリを調べて見つかった、高度に発達したデジタル生物のゲノム0072etqについて述べる。

2.7.7項にゲノム0072etqのコードを示す。このゲノム(0072etq)は二つの特徴を持っている。

1. 新しい自己検査方法

今まで述べてきたデジタル生物は自己検査のとき、コードの始めと終りを示すテンプレートのアドレスを求め、その差からサイズを計算していた。このデジタル生物は前半の36命令のコードだけしか働かない。そしてコード終了テンプレートを使わず、adrf,nop0命令(6,7行目)により、コードの中程のテンプレート(34行目nop1)のアドレスを求め、subCAB命令(10行目)でCXレジスタにコードのサイズの半分(36)をセットし、さらにshl命令(12行目)で2倍して正しいサイズを計算している。

2. ループの展開によるコピー・プロシッジャーの効率化

先祖種のコピーループは以下のような手順により行われていた。

1. 命令語を母セルから娘セルへコピーする。
2. CXレジスタの値(母セルのサイズ)を1減じる。
3. CXレジスタの値が0になったらコピーループを抜ける。0でなければループ内の次の処理を行う。
4. CXレジスタの値(娘セルの命令語のコピー先アドレス)を1増加する。
5. BXレジスタの値(母セルの命令語のコピー元アドレス)を1増加する。

6. コピーループの先頭に戻る。

これに対して0072etqは以下の手順を使う。

1. 命令語を母セルから娘セルへコピーする。(15行目)
2. CXレジスタの値を3減ずる。(16～19行目)
3. AXレジスタの値を1増加する。(20行目)
4. BXレジスタの値を1増加する。(21行目)
5. 命令語を母セルから娘セルへコピーする。(22行目)
6. CXレジスタの値を1減ずる。(23行目)
7. AXレジスタの値を1増加する。(24行目)
8. AXレジスタの値を1増加する。(25行目)
9. 命令語を母セルから娘セルへコピーする。(26行目)
10. CXレジスタの値を1減ずる。(27行目)
11. CXレジスタの最下位ビットを反転する。(28行目)
    ループの最初でCXレジスタの値が偶数ならば、このコードの実行時にはCXレジスタの最下位ビットは必ず1になっている。したがって、この命令はCXレジスタの値をさらに1減ずるのと同じ効果をもたらす。
12. CXレジスタの値が0になったらコピーループを抜ける。(29行目) 0でなければループ内の次の処理を行う。
13. AXレジスタの値を1増加する。(31行目)
14. BXレジスタの値を1増加する。(32行目)
15. コピーループの先頭に戻る。(33行目)

1回のループで3つの命令語をコピーする。これにより、(12)の比較や(15)のコピーループの先頭に戻る処理が、コピーのたびにチェックするのに比べて、1/3で済む。そしてこの3つの命令語のコピーに要する命令実行回数は、数えてみると18であり、これは1命令語のコピーに6命令で済む事を意味する。先祖種の場合では1命令語のコピーに10命令を要しているので40%も効率が向上した事になる。

また、CXレジスタの値はshl命令(12行目)で最初72にセットされていたので、このループは12回まわり、前半の36コードのみが複製される。娘セルの残りの36命令分のスペースはジャンクコードになる。このように必要なスペースの2倍のスペースをとる事により、より多くのCPU時間が得られる。このような現象はとくにタイム・スライスとゲノムのサイズが等しくなるような環境設定を行ったときに現れる。

2.3.8 共生種

実験では確認されていないが、共生関係にある種を考えることもできる。別紙付録のブロック図SymbiosysA,Bは、そのようなものの一例である。共生種Aは自己複製ループを持たず、また共生種Bはコピープロシッジャーをもっていない。そのためどちらも単独では自己複製できないが、この2種が隣あって存在すれば、お互いに不足しているコードを他方から借りることによって自己複製可能になる。

2.3.9 進化による最適化

ここで突然変異率を変えたとき、進化がどのようになるかをゲノムサイズ（コードサイズ）に着目して見てみる。デジタル生物にとっての最適化とは自己複製の効率を上げることである。自己複製の効率化はゲノムサイズを小さくすることで実現できるが、進化した生物はゲノムサイズに対して複製に要する命令実行回数を減らしすことでも効率を上げている。つまり、一命令語のコピーに必要な命令実行回数を減少させて効率を上げている。

Figure:2.3,2.4は500万命令実行の間にゲノムサイズがどのように変わったかを、突然変異率を1,2,4,8,16,32,64,128に設定した場合についての結果である。ここでいう突然変異率とはそれぞれの世代においてN個体につき1個体の割合で突然変異するとした場合のNである。最も突然変異率の高い1と2の場合、すなわちすべての個体か半分の個体が突然変異をする場合、ゲノムは高い突然変異率のために融けてしまうかのようである。デジタル生物たちはシミュレーションの途中で死滅してしまう。突然変異率が4の場合、絶滅のリスクを侵しながらも最も良い最適化を達成した。その他の低い突然変異率8,16,32,64,128の場合の最適化は4の場合より劣っていた。

Figure:2.5は突然変異率を4に固定し、乱数のシードを変えて実験した場合である。断続的に最適化（コードが短くなった）したものもあれば、連続的にスムーズに最適化された場合もある。最も小さなゲノムサイズは22だったが、あるシミュレーションでは27までしか最適化しなかったり、他のある場合では30にとどまってしまう場合もあった。これはシミュレーション実行のたびに局所解に陥ってしまった事を示す。

自己複製アルゴリズムの効率化はサイズの減少以上に進む。先祖種は839回の命令実行回数で自己複製を行う。それに対してサイズ22の生物は146回の命令実行回数で自己複製を行える。これは先祖種に対して5.75倍効率がアップしたことを意味する。**2.7.8項**にこの生物のゲノムを示す。

さらに2.3.7節で説明したゲノム0072etqはさらに効率が上がり、ループの展開を使うことで一命令語あたり6回の命令実行でコピーできる。

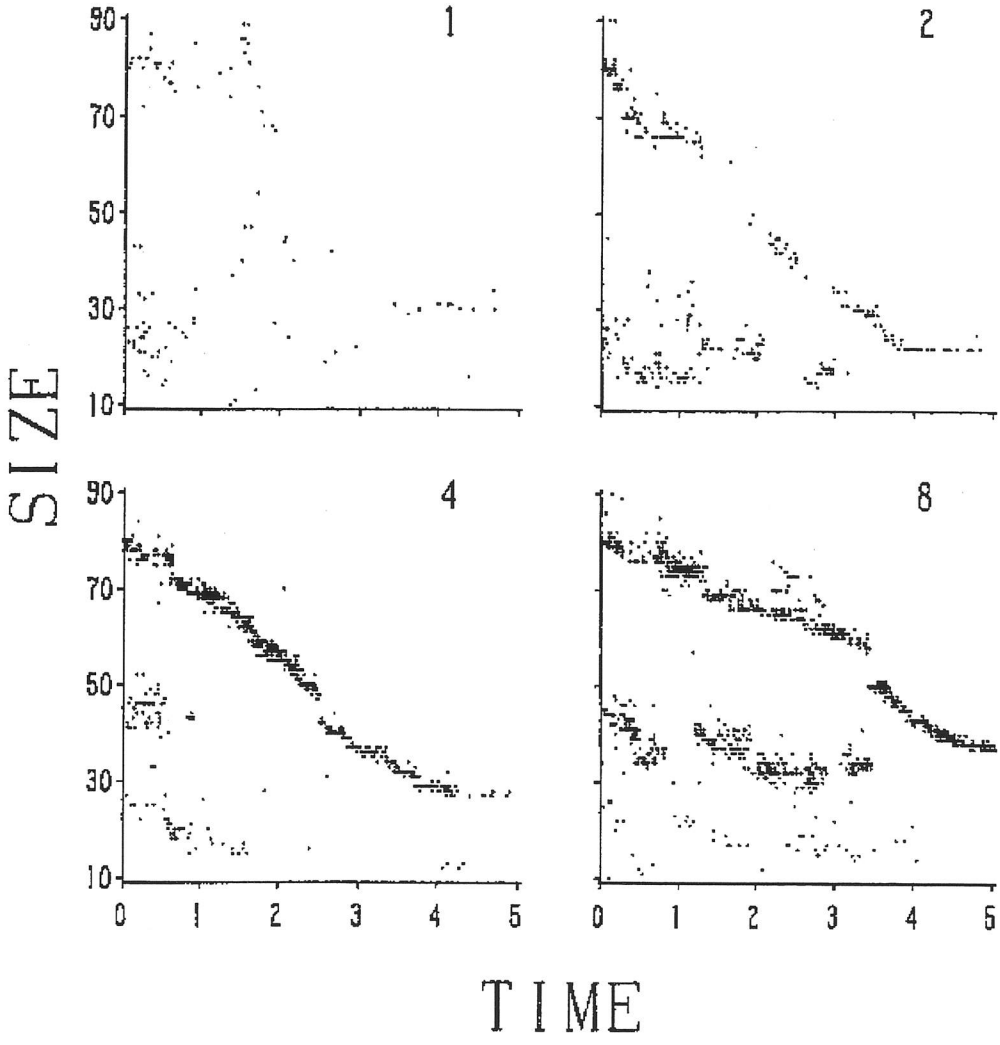


Figure 2.3: 突然変異率を変化させたときの進化上の最適化過程（1）

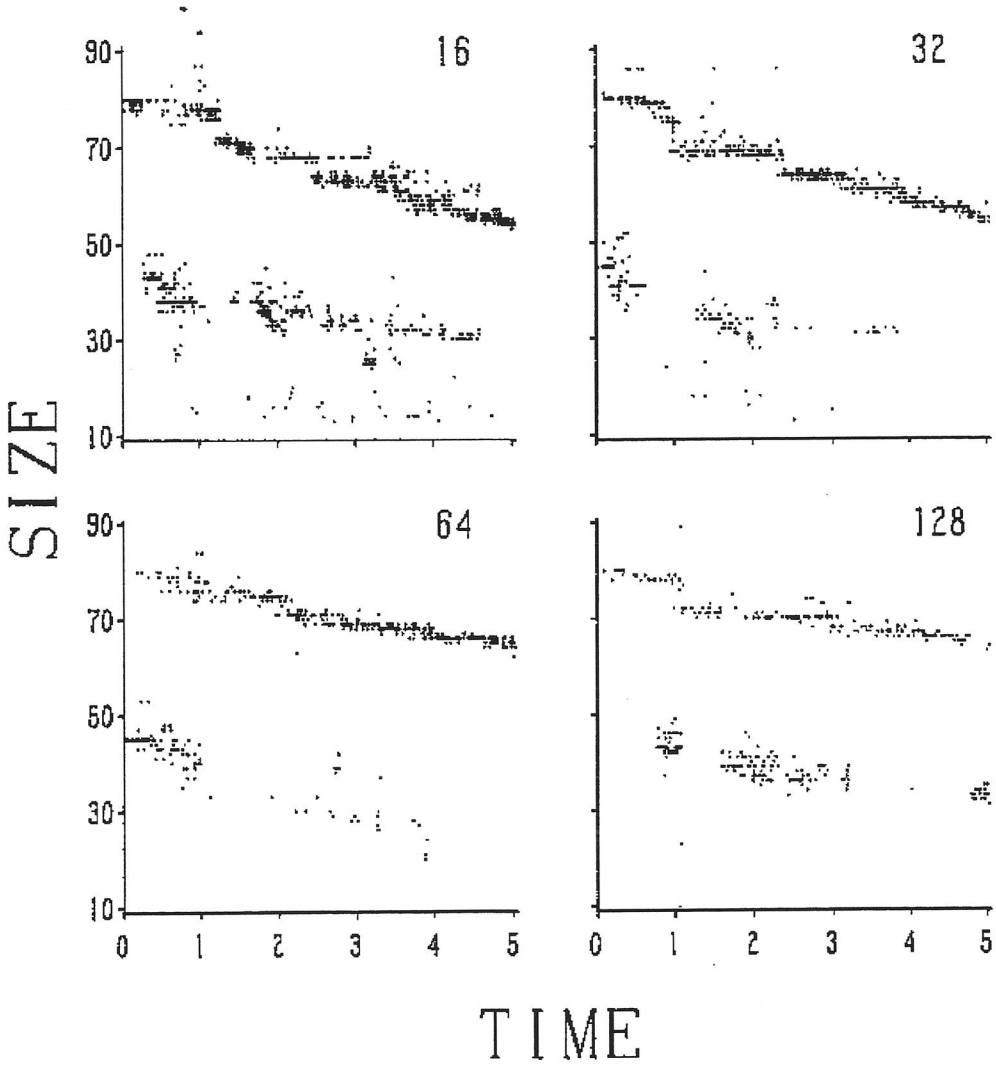


Figure 2.4: 突然変異率を変化させたときの進化上の最適化過程（2）

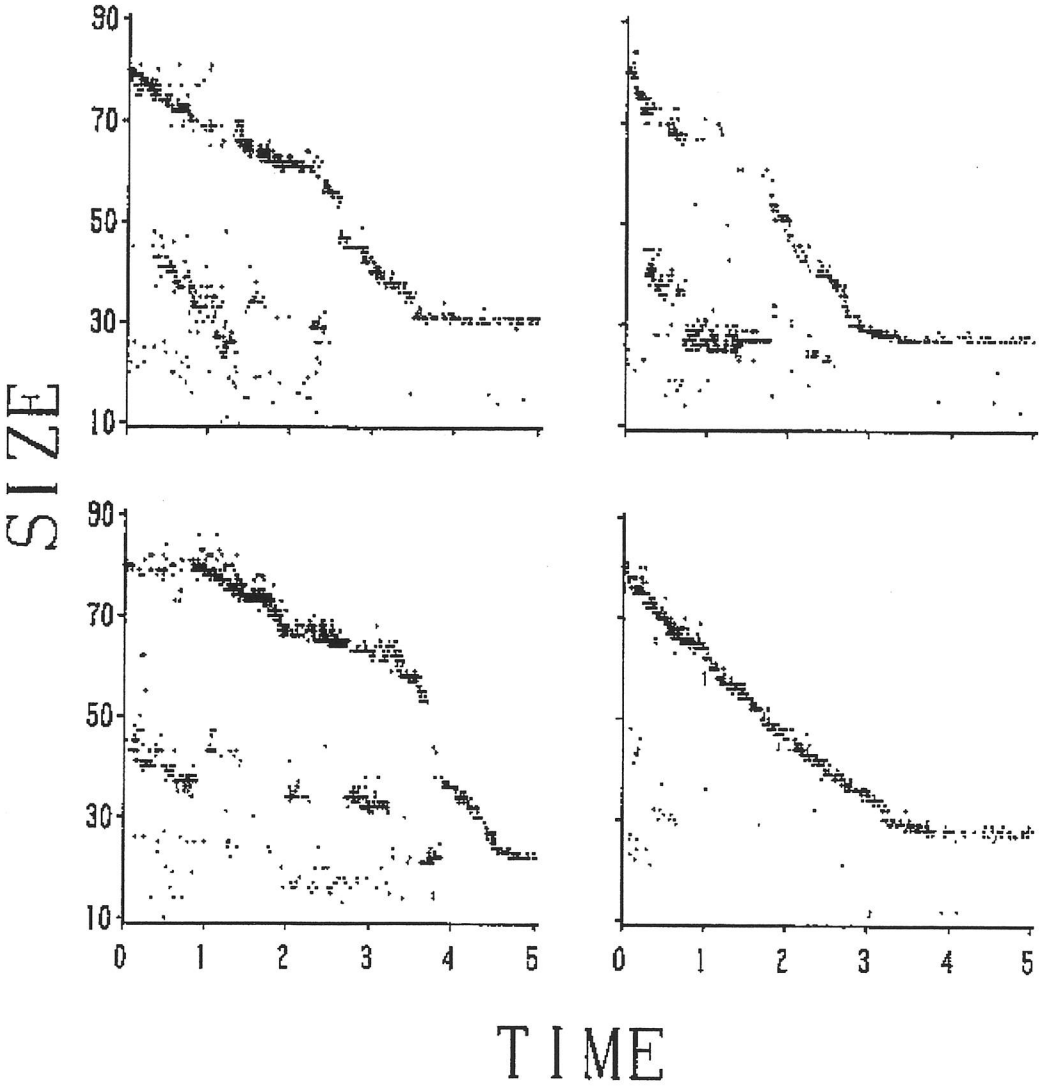


Figure 2.5: 突然変異率が一定（4）で実験を繰り返した場合の最適化のばらつき

2.4 多細胞生物

ここではKurt Thearling(当時Thinking Machines社)とトム・レイが行った、Tierraに多細胞生物の特徴を持たせた並列プログラミングによるデジタル生物の進化について紹介する[Thearling94]。

初期のTierraの仮想生物たちは単細胞生物的な特徴しかなかった。この研究では新たに多細胞的な性質を付け加える事で、**カンブリア爆発**のような進化の多様性を実現しようとした。

ここでは、多細胞性は以下のように機能すると考える。

1. 1) 生物は単細胞からはじまり、細胞分裂により多細胞生物になる。
2. 2) 多細胞生物のそれぞれの細胞は同じゲノム（遺伝子）を持つ。
3. 3) 多細胞生物のそれぞれの細胞の機能の違いは、それぞれの細胞が同じゲノムの別の部分の命令を並列に実行する事により実現される。

実在の有機生命体では細胞それぞれがゲノムをもっているが、Tierraではメモリの節約とコピープロシッジャーに要する時間を短縮するために、細胞は1つのゲノムを共有する事とした。すなわちTierraの多細胞生物はひとつのゲノムとそれを共有する複数の仮想CPUから構成され、ひとつのゲノムに対して複数のCPUが並列に実行される。

多細胞性実現のためにいくつかの新しい命令セットが考案された。

2.4.1 split命令とjoin命令

split命令が実行されると、それを実行した生物に対してもう一つの新しい仮想CPUが用意される。レジスタ、スタック、IPの値は元のCPUから新しいCPUレジスタにコピーされる。ただしDXレジスタの値は異なる。DXレジスタはCPUを区別する値(CPU自己アドレス)の設定に用いられる。split命令のあと両方のCPUのDXレジスタの値は1ビット左シフトし、新しい方のCPUにはさらに1が加えられる。例として以下のコードについて説明する。

```
split   ; 一つのCPUが二つに分裂
split   ; それぞれが更に分裂、四つのCPUができる。
```

はじめDXレジスタの値が二進数表示で0000だとすると、最初のsplit命令により元のCPUのDXレジスタの値は0000のままだが、新に作られたDXレジスタの値は0001になる。それぞれのCPUが2回目のsplit命令を実行すると、オリジナルのDXレジスタの値は0000のまま、オリジナルのCPUから2回目のsplit命令で作られたCPUのDXは0001, 1回目のsplit命令でつくられたCPUのDXは1ビット左シフトして0010, そのCPUが行ったつぎのsplit命令でつくられたCPUのDXは0011となり, これによりすべてのCPUでDXレジスタはユニークな値（**CPU自己アドレス**）をもつ。

join命令はこれを実行したCPUが他のCPUもこの命令に到着するまで、なにもしないで待つ命令である。この命令が実行されたあと、オリジナルのCPU以外はすべて死

んでしまう。これはdivide命令を正常に行うために設けられた。複数のCPUがそれぞれゲノムをある部分のコピーを担当するとしたとき,すべてのゲノムの複製が終ってからdivide命令を行わせるために使用する。

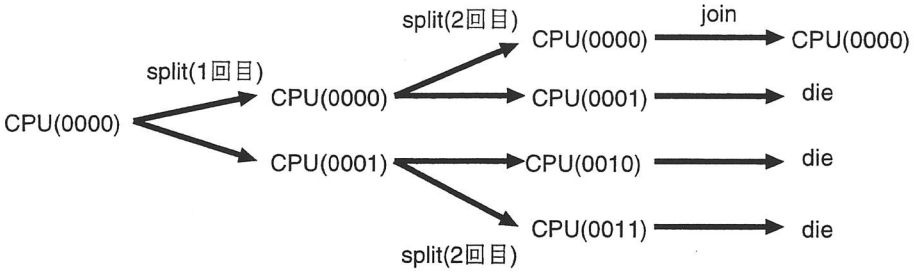


Figure 2.6: Split命令とjoin命令

2.4.2 zeroD命令、shr命令、offAACD命令とoffBBCD命令

split命令はDXレジスタを頻繁に使う。しかし、既存の命令セットにはDXレジスタを直接操作できる命令がなかった。そのために新たにzeroD命令を追加した。zeroD命令はDXレジスタに値0をセットする。split命令の直前にこの命令を置くことでDXレジスタの値を初期化する事ができる。

もうひとつshr命令を追加した。この命令はCXレジスタの値を1ビット右シフトする。CXレジスタには普段ゲノムのサイズが入る。よってこの命令を実行すると、ゲノムのサイズの2分の1の値が得られる。この値はsplit命令によって2倍になったCPUがコピーループを実行するときの繰り返し回数（ゲノムサイズの半分）として利用できる。

各CPUが並列的にそれぞれゲノムのコピー先アドレスを効率的に計算できるよう、"offset"用の命令が追加された。offAACD命令はCXレジスタの値(コピーすべきゲノムサイズ)にDXレジスタの値(CPU自己アドレス)を乗じ、それをAXレジスタ(娘セルの先頭アドレスが格納されている)に加えて、コピー先のアドレスを計算するのに使用できる。offBBCD命令は同様の操作をBXレジスタに行う事によりコピー元のアドレスを計算する。

2.4.3 先祖種

Kurtが考えた先祖種0082aaaのコードを**2.7.9項**に示す。なおここからはレジスタ値の表示はCPUが複数になって繁雑なので省略する。

アルゴリズムは命令セット0の先祖種とほとんど同じである。30025行目のsplit命令の前にzeroD命令が2つ入っているが、このうちの一つはこのコードのサイズが2の倍数になるためのダミーである。2の倍数にする事で2つのCPUが丁度半分づつコードをコピーできるようにした。

なお、この並列CPU版のTierraはコネクションマシンCM-5(米国シンキングマシンズ社製[Kimezawa94参照])を用いて開発され実行さコード終了アドレスれた。

2.4.4 進化

この並列CPU版のTierraを実行してしばらくすると、シングルCPU版のTierraと同様にパラサイトが出現したり、テンプレートサイズが減少してコードが短くなっていく最適化などの現象が起った。ところが、1つの娘セルを複製するのに要する実行回数を種ごとにプロットしたグラフをみると、ある命令実行回数(215ミリオン命令)頃に急激な効率の上昇がみられた。これはsplit命令の実行を増やして短時間で自己複製が行えるように進化した生物が現れた事を物語っている。実際にコードを調べてみると、それまでもっと多くスープを支配していた種のサイズが44だったのに対し、新たに支配的になった種のサイズは52に増えていた。にもかかわらず複製効率はアップしていた。これは増えたコードのうちの11コードが意味のないコード（**イントロン**)で、split命令をひとつ増やしたときコードサイズが4の倍数になるようにわざと含まれたたものなのであろう。その後イントロンはすこしずつ減少し、Tierra実行開始から1億命令後にはサイズは40になった。このゲノム(0040aba)のプログラム・コードを**2.7.10項**に示す。

このコードを見て気づくのは、まだイントロンが3つ残っている事である。これらを取り除く事はできない。なぜなら4CPUのコードは4の倍数でなければ正しく娘セルの複製ができないからである。

ここで説明した多細胞版Tierraはある程度の多細胞性に成功している。しかしトム・レイが目指しているカンブリア爆発という現象までは起きていないようだ。トム・レイはデジタル生物の多様性を増すためには、デジタル環境そのものを複雑にしてやる事が必要だと考えている。そして現在、Tierraをインターネットに接続し、複雑なトポロジーを持ったネットワーク環境下で、デジタル生命の進化を起そうとしている。

2.5 Network Tierra 1 (なぜ Network Tierra か)

Tierraの全体的な環境を左右するのはCPUの能力とスープメモリのサイズである。CPUの能力が高い程、短時間で多くの進化が起り、スープメモリのサイズが大きい程、より多くのデジタル生物たちが生息する事ができる。しかし、1台のコンピュータの内部だけではこれら以外に環境を特徴づける事ができない。デジタル生物の多様性を増大させるための、これまでの多くの試みはどれも大きな目標(カンブリア爆発)には到達しなかった。

最大の障害はTierraを実行してしばらくすると、非常に簡単な20ないし40くらいのサイズのデジタル生物たちがスープメモリを支配してしまう事である。これらの生物たちは基本的に自己複製以外の能力を持たない。そしてTierraではより効率的に自己複製を行える種が有利であるため、もっと複雑な機能をもったサイズの大きな種の出現が妨げられてしまうのである。

このような問題を解決するための新な試みが始まっている。Tierraを実行しているコンピュータをネットワークで結び、そのネットワークの中でデジタル生物の進化を実現させようというものである。ネットワーク、それもインターネットのような巨大なネットワークはTierraシステムの環境を格段に複雑にする。

2.5.1 空間トポロジー

ここでTierraの“空間トポロジー”について考察してみよう(ここでの説明は文献[Ray95a]のSpatial Topologyの章によってる。)。デジタル生物はコンピュータのメモリ空間、主としてRAMメモリに生息するが、それらはディスクやその他の記憶装置、そしてネットワークの向うにも生息しているとしよう。デジタル生物が存在している空間概念は我々有機生物が存在している空間とは全く異なる。

現在ほとんどのPCやワークステーションは数十Mbyte以上のRAMメモリを持っている。これは一様なメモリで、メモリ内のどの場所も数値アドレスでアクセスできる。そしてこの数値アドレスは連続した整数値で表せる。このことにより我々はメモリ空間は直線的な一次元空間のようなものだと考えるだろう。

しかし、この1次元空間の概念は必ずしも正確ではない。メモリの空間トポロジーを理解するには「メモリ内の2つの場所の距離とは何か」を問う事が必要である。その距離をアドレス値の差として考えるべきではない。最も適当な単位は、2つのアドレスの間で情報を移動させるのに要する時間を測定する事である。

RAMメモリ内の情報は、ある場所からある場所へと直接動かす事はできない。情報は一旦CPUのレジスタに送られ、そこから目的のメモリアドレスに転送される。このようにRAM内の2つの位置の距離の差は、情報がRAMからCPUへ転送される時間と考えられ、メモリ内のどのRAMからもCPUへ情報を転送する時間は同じなので、メモリのアドレス値がどんなに異なっていようと、2点間の距離は等しいと考えられる。

すべての点の間の距離が等しいという空間はユークリッド空間ではない。しかしながらメモリの通常のアドレスの付け方では、局所的には一次元と考えることができる。コードがCPUによって実行されるとき命令ポインタはジャンプ命令がない場合は順番

にインクリメントされる。順番に命令が実行される場合は、メモリは一次元空間であると考えてよいであろう。Tierraの相補的テンプレートの探索などのようなサーチ命令はメモリを、探索範囲内に限って一次元空間として扱った例である。しかしこの場合もメモリー全体としては決して一様な一次元空間ではない。

我々は3次元空間に住んでいるため、コンピュータメモリ内にも我々になじみやすい空間トポロジーを仮定してしまう傾向がある。これはメモリが次元ユークリッド空間であるという間違った認識と、Tierraのデジタル空間をユークリッドメモリ空間の次元を増加する事によって拡大させようという誤った認識を招く。

Tierraの空間モデルを拡張しようという努力のほとんどが、スープメモリの次元をユークリッド空間的に2次元や3次元に拡張しようとするものであった。たしかに、我々の住んでいる3次元空間を、もし2次元や1次元の世界に縮退させてしまったら、我々のような生命は誕生して来なかっただろう。しかし、この考えの欠陥はコンピュータメモリが一次元ユークリッド空間でできているという仮定から起っている。

トム・レイが提唱する生命合成の基本的アプローチは、それぞれのメディアの性質を尊重し、そのメディアの中の生命の自然な形を発見せよという事である。デジタルメディアの中に有機生命のような形態を無理に仮定してはならない。

サイバースペースの構造を少し考える事でメモリ・トポロジーの多様性を増すヒントが得られる。Intel 80x86で使用されるような有名なセグメントメモリを考えよう。このデザインでは64Kバイト毎にメモリブロックを選択する。ブロック内でのメモリはどれも等時間でアクセスできる。しかしブロック外のメモリへのアクセスは2倍遅くなる。

キャッシュメモリはRAMよりもすばやくアクセスできる。キャッシュメモリはRAMよりも近いと考えられる。ディスクへのアクセス時間はRAMよりも遅い。したがってディスクはRAMよりも遠い。CPUのレジスターは小さなメモリしかないがそのアクセススピードは最も高速である。同一CPU内の各レジスタはお互いに最も近い距離にあると考えられる。

ネットワークコンピュータシステムでは、情報はネットワークを介して行われ、アクセス時間のほとんどは情報の転送時間である。これはいままでに説明したものの中で、もっとも遅い。すなわち遠い距離にあると考えられる。またネットワーク自体、LAN,WAN,インターネットと極めて複雑なトポロジーを有している。

これら,レジスタ、キャッシュ、RAM、ディスクそしてネットワークを考慮する事により非常に複雑なトポロジーをが得られる。この複雑なトポロジーの性質はデジタル生物の進化に大きく寄与するに違いない。

2.5.2 仮想ネットワーク

ネットワーク版Tierraは**TCP/IP**によりお互いに通信を行う。TCP/IPはインターネットで使用されている最も基本的な通信プロトコルである。ネットワーク内の各ノードはマップファイル[Charrel95]を参照する事により、どのノードと通信できるかわかるようになっている。Tierraはこのマップファイルで参照できるノード以外には通信しないので、関係ないコンピュータに情報が送信される事はない。Network版Tierraではこの

ように特定の参加マシンだけでネットワークを構築する。Tom Rayはこれを仮想ネットワークと呼んでいる。Tierraが送受信するメッセージはコンピュータから見た場合、単なる文字列にすぎないので、コンピュータ・ウイルスやインターネットワームのように、ネットワークやコンピュータに悪影響を与える恐れはない。セキュリティに関しては1995年Santa Fe研究所で行われたFirst Tierra Workshop[TierraWorkShop95]で議論された。

TPing

現在、ネットワーク版Tierraのための命令セットを作成中である。ある命令セットはゲノムの全コードを別のノードに転送する機能をもっている。この命令セットによりデジタル生物はより生存に適したノードに移動する事ができる。

別の命令**TPing**はデジタル生物にとってセンサーのような働きをする。この命令は別のノードの大域的な状態を問い合わせるものである。口絵3はTPing命令により得られたデータを**VRML**により表示したものである。これらの図のデータは日本(ATR)、米国(University of Delaware, Santa Fe Institute)、ベルギー(Free University of Brusells), スイス(Swiss Federal Institute in Lausanne)などの研究機関のマシン約100台を使って行った予備実験の結果を表示している。

口絵3の上の図には赤い球体、青い球体、緑の球体がそれぞれいくつかづつ描かれている。青い球体は仮想ネットワーク上の各ノードのCPUの処理スピードを表している。赤い球体は各ノードのスープサイズを表している。緑の球体は各ノードに生息しているデジタル生物の数を表している。緑の球体1個について100匹の生物が生息している事を意味している。この図はそれを観測している生物にとってごく近くのノードの様子である。

口絵3の下の図はTPingで得られたデータのほぼ全体を表したものである。

全体としてみると、各ノードは円錐上の側表面に散らばっている。頂点部分がこの上の図で拡大して見えていた部分で、この部分にあるノードこのデジタル生物にとって最も近い距離にあるノードである(おそらく同一LAN内のノードであろう)。円錐の頂点から遠いほどデジタル生物とって遠いノード(おそらくインターネットを経由した海外のノード)である。

このTPing命令により、はたしてどのようなデジタル生物が出現できるだろうか。ネットワーク版のTierraは一番低いプライオリティのプロセスとして実行される。したがって、コンピュータの使用者が仕事などのために他のソフトウエアを利用しているときには、Tierraはほとんど停止状態になっている。しかしインターネットで結ばれる事によりデジタル生物はCPUタイムとメモリ空間を求めてインターネット内を移動することができる。インターネット内のコンピュータの稼働率に日時変動があれば(例えば昼間は高く、夜間は低いなど)、デジタル生物は稼働率の低い(すなわちCPUタイムとメモリに余裕のある)ノードに移動するようになるだろう。そしてさらに進化してあらかじめ変動が起るまえに有利なノードに移動するような賢い種が現れるかもしれない。

他にも様々な進化が考えられる。しかし、有効な命令セットの設計には今のところまだまだ多くの検討が必要な段階である。

2.5.3 COMMITMENT

ネットワーク版Tierraはまだ開発途上で、現在有効な命令セットの設計およびプログラミング、そしてそれを用いた先祖種の設計作業が行われている。ネットワーク版Tierraでは最終的にはコンピュータとインターネットを利用できる数千名のボランティアを募り、大規模な実験を行う予定である。(現在数百ノードのレベルでの予備実験を行っている。)参加可能なプラットフォームとしてUNIX,Windows,Macその他の各OSが実行できるマシンを想定している。現在の開発作業は主としてUNIXマシンで行われているのでWindowsやMacへの移植作業の終了は若干遅れるだろう。

ネットワーク版Tierraが本格的に動き出すと、これは全世界的な「デジタル」生態系とみなす事ができる。本当の熱帯雨林のようにデジタル生態系では多種多様なデジタル生物が繁栄するかもしれない。そうすると、それらの生物たちを観察する「デジタル生態学者」が現れるだろう。またデジタル生態系から採取したゲノムを自分のマシンへ移植して品種改良し、人間に役に立つソフトウエアを開発するような仕事もできるかもしれない。

2.6 Network Tierra 2 (ネットワーク型多細胞生物)

本節では1998年5月現在のNetwork Tierraの研究成果[Ray98]について解説する。

2.6.1 ゲノム・遺伝子・細胞の分化と組織

ここで、Network Tierraが実現を目指す機能について説明する。そのために、まず多細胞有機生物における遺伝子の分化について述べる。有機生物における「ゲノム」はDNAの連鎖であり、それぞれの細胞はこのゲノムの完全なコピーを持っている。またゲノムは多くの「遺伝子」からなっている。遺伝子は特徴的な機能、主に蛋白質を合成する機能を有するDNAのセグメントである。細胞は皆同一のゲノムを持っているにもかかわらず、通常その一部の機能しか表に現さない。遺伝子によって表現される機能は、「細胞タイプ」を決定する。同じ細胞タイプが集まったグループの事を「組織」と言う。異なった組織は、それぞれ表出する機能が異なる遺伝子のサブ集合による「分化」した細胞によって構成される。

たとえば、血液細胞は全能性幹細胞を出発点として、骨髄性幹細胞とリンパ球系幹細胞に分化する。骨髄性幹細胞はまた赤血球幹細胞を経て赤血球に、顆粒球系幹細胞を経て好中球とマクロファージに、好酸球系幹細胞を経て好酸球に、巨核球系幹細胞を経て血小板に分化してそれぞれの機能を果たす。リンパ球系幹細胞もB細胞とT細胞に分化して免疫システムを司る。もともと、同じ遺伝子を持った同じ細胞が分化により異なる機能を持った細胞に変わり、かつお互いが協力して組織を構成し一つの個体を作り上げるのである。

Tierraのデジタル生物は、自己複製するコンピュータの機械語コードでできている。そして多細胞性はそれぞれの細胞ごとに仮想CPUを持ったマルチスレッドにより実現される。すなわち、一本のプログラムが複数のプロセスとして実行される。有機生物

では、ゲノムは各細胞別々に有していたが、Tierraのデジタル生物は同じ一本のゲノムを共有する。というのはコンピュータのメモリーには、我々の住んでいる世界のような空間的な概念が存在しない。仮想CPUはメモリ空間のどのアドレスにもほぼ等しい速さでアクセスできる。したがって、デジタル生物は多細胞性を実現するために、細胞毎にゲノムのコピーを持つという無駄・冗長さを省くことができる。

デジタル生物のゲノム、すなわち機械語コードは、ある機能を実現するサブルーチンの集まりから構成され、それぞれのサブルーチンは有機生物の遺伝子に対応する。スレッド(仮想CPU)は同じ一本のプログラムのうち、それぞれ特定のサブ集合（遺伝子）を実行する。特定の遺伝子を実行するスレッドはその細胞のタイプを決める。同じ細胞タイプをもたらすグループは組織を構成する。異なる組織はゲノムのアルゴリズムの異なる遺伝子を実行する。

Network Tierraは、このように機能が分化した組織によって構成されるデジタル生物とその進化を実現しようとしている。

2.6.2 先祖種

マルチスレッド型デジタル生物はネットワーク環境に生息し、そこではコンピュータ・リソース（おもにCPU時間）が淘汰要因を左右する。デジタル生物はネットワーク内のさまざまなマシンの状態を知るためのセンサーと、それから得られたデータを処理する機構を用いて、ネットワーク内を移住(migration)する。

実験は、すでに2つのタイプに分化した組織を持った**Figure:2.7**のようなマルチ・スレッド型デジタル生物のプログラム(先祖種)を走らせる事から始める。この先祖種は320バイトの機械語コードを持つが、どのスレッドにもすべてのコードを実行するというものはない。ゲノムは、便宜上名付けられた6つの部分(遺伝子)によりできている。そしてそれらのいくつかは、さらに小さな機能に解剖する事ができる。遺伝子のうちの2つは単細胞から成熟した10個の細胞を持った多細胞生物に成長するためのもの(sel遺伝子、dif遺伝子)である。ある一つの遺伝子(rep遺伝子）は自己複製組織によって実行され、また別の遺伝子（sen遺伝子）はセンサー組織により実行される。残りの2つの遺伝子は自己複製組織とセンサー組織両方により実行される（cop遺伝子、dev遺伝子）。

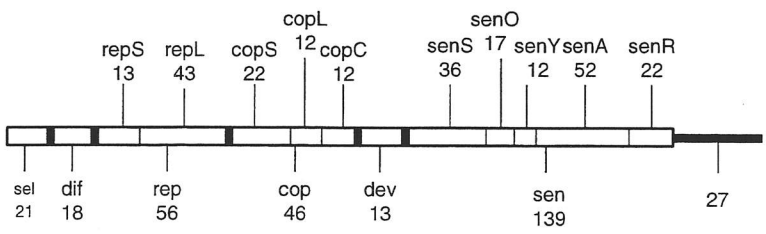


Figure 2.7: Network Ancestor

先祖種は最初 1 個の細胞から始まり、細胞分裂（スレッドの生成＝新しい仮想 CPU の生成）により成長する（下図 Developmental Pattern 参照）。

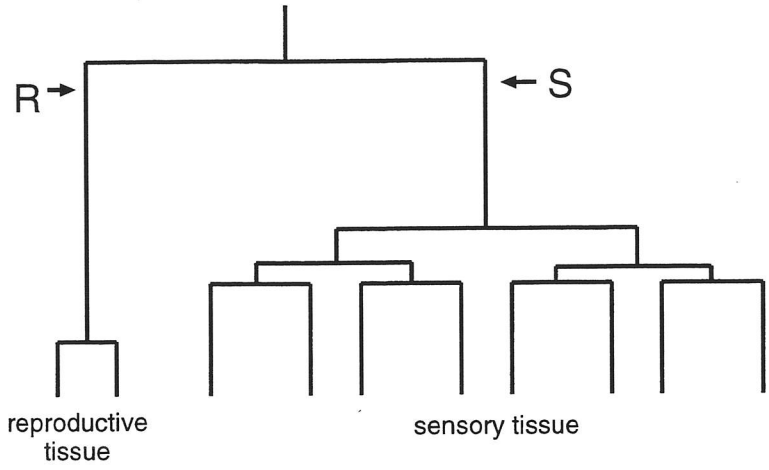


Figure 2.8: Developmental Pattern

まず 1 個から 2 個に分裂し、それぞれの細胞が分化する。すなわち，条件分岐命令および jump 命令により、それぞれ別の機械語コードを実行するようになる。一方の細胞は自己複製細胞に、もう一方はセンサー細胞に分化する。さらに自己複製細胞はもう一度分裂して、2 個の自己複製細胞でできた自己複製組織になる。自己複製細胞は自分のコピーを作成し、divide 命令により娘デジタル生物を生成し終わると、2 つのうち 1 つが停止 (halt) し、その後また分裂しなおして自己複製プロセスを再開する。

センサー細胞は 3 回の分裂により 8 個の細胞になってセンサー組織をつくる。これらのセンサー細胞は生まれると getipp 命令により、後で説明する Tping データを取得し、それをデータエリアに書き込む。一度このような成熟したデジタル生物になると、後で述べる移動すべきネットワーク内のコンピュータを決定するためのセンサー・プロセスを実行する。センサー・プロセスの過程で 8 つのスレッドのうち 7 つは停止する。残った一つのセンサー・セルは再び分裂（スレッドを生成）し、Tping データを取得する。

2.6.3 センサー・システム

ここでセンサー・システムが使用する Tping データについて説明する。Tping データは C 言語の構造体データとして以下のように定義されている。（ここで I32s は 32bit の符号付き整数型データ、I32u は 32bit の符号なし整数型データタイプである。）

```
struct TPingData        /* Tping メッセージの構造 */
{       I32s t;              /* メッセージタイプのタグ */
        I32u address.node;  /* ノードの IP アドレス */
        I32u address.portnb;/* ソケットのポート番号 */
        I32s cellID;        /* 一つのノード内におけるデジタル生物の個体識別番号 */
        I32s ranID;         /* ネットワークに全体での個体識別番号 */
        I32s FecundityAvg;  /* デジタル生物が死ぬまでに生む娘セルの平均値 */
        I32s Speed;         /* 毎秒当たりの平均命令実行数 */
        I32s NumCells;      /* 一つのノードにおけるデジタル生物の数 */
        I32s AgeAvg;        /* 死ぬまでの平均命令実行数 */
```

```
        I32s SoupSize;       /* Tierra スープ（メモリ）のサイズ */
        I32u TransitTime;    /* ネットワークの伝達時間(milli seconds) */
        I32u Fresh;          /* このデータの最終更新時刻 */
        I32u Time;           /* ノードの時刻 */
        I32s InstExec;       /* このTierraサーバーで実行された全命令数 */
        I32s InstExecConnect;/* ネットワークに接続されている間に実行された命令数 */
        I32s OS;             /* OSタイプの識別タグ */
};
```

- **address.node**（IPアドレス）によって、デジタル生物は移動したいマシンのアドレスを知る事ができる。
- **FecundityAvg**は直近の100万命令実行の間に、そのノードにいたデジタル生物たちが、死ぬまでまたは他のノードに移住するまでに生んだ子孫の数の平均である。
- **Speed**は直近の100万命令実行の間の毎秒当たりの平均命令実行回数である。
- **NumCells**は、このデータが作成された時点における、そのノードに生息しているデジタル生物の総数である。
- **AgeAvg**は直近の100万命令実行された間に生きていたデジタル生物の平均寿命(命令実行回数)である。
- **InstExec**はそのノードでTierraが実行開始されてからの経過時間(命令実行回数)である。
- **InstExecConnect**はネットワーク接続後に経過した時間であるが、今回の実験では常にネットワークに接続されているので、**InstExec**と同じ値になる。

それぞれのTierraは「マップ・ファイル」というNetwork Tierraを構成する各マシン（ノード）のリストを持っており、TPing Dataをそこに格納する。最初デジタル生物が生まれると、乱数によってこのリストのあるノードを指定する。デジタル生物がgetipp命令を実行することにより、ひとつのTpingデータがCPUレジスタで指定されたスープメモリのあるアドレス（データエリア）に書き込まれ、リストのポインターがインクリメントされる。

2.6.4 センサー・プロセス

先祖種は上記のTpingデータを保存しておくための、512バイトのデータエリアを持っている。8個の細胞（スレッド）はそれぞれ64バイトのTpingデータを8つに分けられたデータエリアのうちのひとつに格納する。そしてセンサーアルゴリズムによって移動すべきノードを決定する。センサーアルゴリズムは, 3つの段階を踏んでノードを決定をする。まず、隣り合う4つのペア（すなわち、1と2、3と4、5と6、7と8番目のデータエリア同士でペアを作る。）でTpingデータを比較し、優れたデータと判断した方を左側のデータエリア（すなわち、1、3、5、7番目のデータエリア）にコピーする。

第2段階では、1番目と3番目、5番目と7番目のデータエリア同士を比較し、良いと判断したものをそれぞれ、1番目と5番目にコピーする。第3段階で、1番目と5番目のデータエリアを比較し、良いと判断したものを1番目のデータエリア(データエリア先頭番地)に格納する。自己複製アルゴリズムは娘細胞の移動ノードを、データエリアの先頭番地に書かれたIPアドレスと認識して、そのノードに娘細胞を送り出す。

このプロセスは以後このデジタル生物が死ぬまで続くが、2度目のプロセスでは8つのセンサー細胞のうち7つのセンサー細胞だけが、データエリアにTPingデータを格納する。よって、最初に選択された最良ノードの情報は保存される。

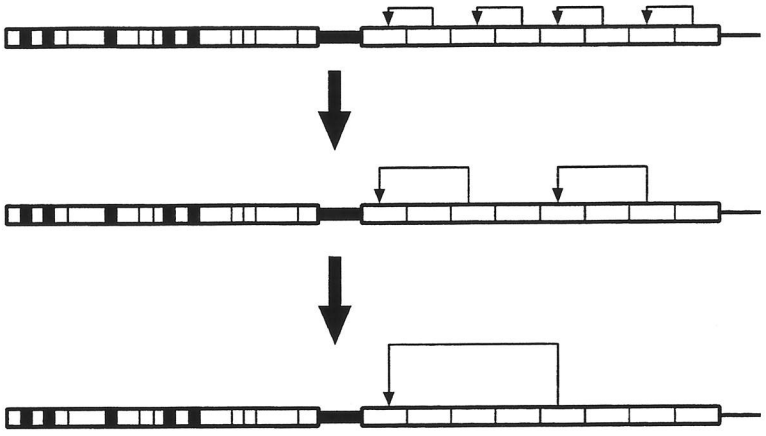


Figure 2.9: Sensory Processing

2.6.5 遺伝操作

Kozaの遺伝的プログラミング(Genetic Programing)[Koza92]やKarl SimsのGenetic Image[Sims92]は**Lisp言語**に突然変異や、**交叉**を行う事で非常に高いレベルの進化を実現した。一方、Tierraではこれまで、直線状の命令語列にビット反転を施す事以外の遺伝的操作は行ってこなかった。より進化の速さを高めるため、Tierraでも交叉や**挿入**、**欠失**など他の遺伝操作を加える事とした。また、突然変異にもビット反転以外に、ある命令語を別の全く異なる命令語と取り替えるような変異の仕方も採用した。これら新しい遺伝操作は、新たなゲノムが生まれる場合32回につき1回の確率で行われるものとした。

2.6.6 ネットワークの構成

Network Tierraはちょうど World Wide Webのように構成される。インターネットに接続されたマシンにTierraサーバーを置き、デジタル生物が自由に移動できるようにする。Tierraサーバーが走っていないマシンにデジタル生物が移動する事はない。またTierraサーバーはTierra browsers(**Beagle**)[Kimezawa96]によって、その内部を観測する事ができる。BeagleもTierraサーバーが走っていないマシンにはアクセスする事はできない。

複数のTierraサーバーが接続される事によりインターネットの中にサブネットを構成する。デジタル生物はTierraサーバーを介して得られるCPUサイクルの獲得とRAMメモリーへのアクセス権を得る。ここでもう一度注意したいのだがデジタル生物は決して、Tierraサーバーが許可した以外のRAMやDiskにアクセスする事はない。

デジタル生物たちは、CPU時間を求めてネットワーク内を移動するための戦略を獲得する事を求められ、これが淘汰の最も大きな要因となる。本実験ではより複雑な移住戦略の進化を促すため「大変動（大量絶滅）」(apocalypse)の機構を取り入れた。すなわち全く移住せず同じマシンに居座っていようとするデジタル生物を完全に死滅させるために、ランダムな周期でネットワーク内のあるマシン内のすべてのデジタル生物に死を与えるのである。これにより、生き残れるのは少しでも移住能力のあるものに限られる。

また、Tierraサーバーの優先度は最低の値(**Nice値**=39)に設定される。これによりTierraサーバーがデジタル生物たちに与えられるCPU時間は、そのコンピュータを利用しているユーザーの行動に左右される。もしユーザーがTierraサーバー以外のソフトウエアを実行すると、デジタル生物が利用できるCPU時間が減少する。また、ユーザーがキーボードやマウスを操作するとTierraサーバーはそれから10分間、スリープする。このようなヒトによる動的な環境変動により、細胞分化とその進化を促してくれる事を期待している。

本実験はローカルエリア(ATR社内LAN)に接続された60台のUNIXをOSとするSun Workstationで行われた。

2.6.7 結果

成長パターンの進化

先祖種では2つの自己複製細胞が、ゲノムのコピーを担当するが、進化の結果、自己複製細胞が増殖し8つずつの2組になって自己複製を行う種や、16、32、64、256個の細胞がそれぞれ組織を作って自己複製を行うような種が現れた。

またセンサー細胞（組織）も同様、8つのセンサー細胞で作られる組織が複数に増えたり、16個や32個のセンサー細胞で一つのセンサー組織を形成するようになった種が出現した。

センサープロセス

TPingデータの処理方法についても調査した。先祖種には以下のようにTpingデータの処理方法が違う3つの種を用意しておいた。

- **960aad**は**Speed/NumCells**を比較しその値が大きい方のノードを選択する。
- **960aae**は**Fucundity**の大きい方を選択する。
- **960aaf**は**Fecundity**×**Speed**が大きいノードを選択する。

実験の最初の段階では、これら 3 つの種のうち、どれか一つが全体を支配するような現象が起こった。ある実験では Speed/NumCells が、別の実験では Fecundify × Speed を用いたアルゴリズムが支配的になった。さらに、いくらか進化か続くと Tping データを計算する代わりに定数を比較に用いるような種も現れた。例えば、Speed が 256 より小さければ、右側のデータエリアのデータを左側のデータエリアにコピーするなどである。

遺伝子の変化

遺伝子の変化は遺伝子の種類によって異なった。成長するための遺伝子 dif,dev はほとんど変化しなかったが、データをコピーする copL 遺伝子は非常に大きく変化した。copL 遺伝子はオリジナルの Tierra 以上のループの展開 (unrolling loop) を達成した。これはデータのコピー効率が非常に大きな淘汰圧となって働いた事を物語っている。

遺伝子重複

重複は新しい組織が出現するための鍵となる現象であるはずである。本実験では cop 遺伝子が重複し、一方は自己複製組織に、もう一方はセンサー組織専用に使われるように進化した場合が観察された。これは遺伝子の多様性の増加の初期段階と思われる。

「群衆」行動

ある特定の種がネットワーク全体に占める割合が高くなると、ほとんどのデジタル生物が同じ選択を行い、結果として特定のノードに集合してしまうという「群衆」(mob) 行動が起こってしまった。試みに、Speed/NumCells をノード選択の基準とした先祖種を 4 台のマシンで構成されたネットワークで (遺伝オペレーターは停止させたまま) 実験してみた。そして 4 つノードのうち 1 台は突出して Speed/NumCells の値が高いように設定した。すると、すべてのノードにおいて、生まれた娘デジタル生物はすべてこの最も値が高いマシンに集中しようとした。そして他の 3 台のノードでへの移動がなくなり、娘デジタル生物も生まれなくなってしまった。その結果、これら 3 つのノードでは先祖種が永遠に生き残り、高い fecundity 値を得る事になった。

一方、移動が集中したマシンでは絶えず新たなデジタル生物が生まれたり、他のノードから流入してくるので、ちょっとでも古くなったデジタル生物はたちまち、リーパーによって除去され、fecundity はゼロに近くなった。

このような現象は、大きなネットワークにする事で多少改善される。というのはノード数が多いとデジタル生物はすべてのノードを比較するのが難しくなるので、特定のノードに集中してしまう現象を緩和できる。また、進化により新たなアルゴリズムが開発され、特定マシンへの集中という現象を回避できる。256 < =Speed を使うアルゴリズムは一見賢そうではないが、マシンの選択を曖昧にするのでそれなりに効果がある。

センサーシステムの退化

いくつかの実験では、センサーシステムが退化してなくなってしまう現象が観察された。これはセンサーシステムを発達させるための淘汰圧が緩和されたせいではないかと思われる。本実験はATRのローカル・エリア・ネットワーク内で行われた。ATRでの就業時間の関係で、ノードの負荷は時間的に変化する。進化の早い段階で、センサーシステムを進化させるためのある重要な淘汰圧が働いていなければ、以降、センサーシステムは退化してしまうのではと考えられる。

2.6.8 この章の結び

本研究には2つの大きな目標があった。一つは分化した細胞をもったデジタル生物たちが、進化の長い期間にわたって存続し続けること。もう一つは、たくさんの種類の分化が起こる事である。

今回は一つ目の目標は達成された。先祖種は2種類に分化した細胞による組織によって構成され、その組織形態や組織内での各プロセス（自己複製やセンサープロセス）が進化した。

2つ目の目標が達成される兆候は今回は得られなかった。しかし遺伝子重複はこの現象を起こしてくれる可能性を秘めていると思われる。

デジタル生物が生存を続ける最低用件は、自己複製ができる事である。先祖種のセンサーに関する機械語コードは157命令であり、自己複製のコード136命令よりも多い。しかもセンサーはゲノムサイズの約2倍のデータ領域を必要とする。そしてセンサープロセス自身が多くのCPU時間を消費する。しかし、ネットワーク環境はこのような複雑なシステムを維持するに足る淘汰圧をもたらした。これは、Tierraをネットワーク環境に拡大する事により、複雑性を高めるという方法が妥当であるということの理由と言える。

デジタル生物の移住パターンはそれ自身が、ネットワークの重要な適合度地形を作る。先祖種のアルゴリズムは群衆行動を起こしてしまうが、進化によるアルゴリズムの変化によってこの問題は解決される。

2.7 命令セット一覧表と先祖種および進化した種のコード

2.7.1 命令セット 0 の一覧表

16 進	2 進	ニーモニック	説明
No Operations: 2			
0x00	00000	nop0	no operation テンプレートを作る
0x01	00001	nop1	no operation テンプレートを作る
Calculation: 9			
0x02	00010	not0	CX の値の最下位ビットを反転
0x03	00011	shl	CX の値を左シフト, CX $<<$ 1
0x04	00100	zero	CX の値を 0 にセット, CX = 0
0x05	00101	ifz	もし CX==0 なら次の命令を実行、そうでないらなら次の命令をスキップ
0x06	00110	subCAB	引き算 CX = AX - BC
0x07	00111	subAAC	引き算, AX = AX - CX
0x08	01000	incA	インクリメント AX, AX = AX + 1
0x09	01001	incB	インクリメント BX, BX = BX + 1
0x0A	01010	decC	デクリメント CX, CX = CX - 1
0x0B	01011	incC	インクリメント CX, CX = CX + 1
Memory Movement: 11			
0x0C	01100	pushA	AX の値をスタックに入れる
0x0D	01101	pushB	BX の値をスタックに入れる
0x0E	01110	pushC	CX の値をスタックに入れる
0x0F	01111	pushD	DX の値をスタックに入れる
0x10	10000	popA	AX の値をスタックから取り出す
0x11	10001	popB	BX の値をスタックから取り出す
0x12	10010	popC	CX の値をスタックから取り出す
0x13	10011	popD	DX の値をスタックから取り出す
0x14	10100	movDC	DX に CX の値を代入 DX = CX
0x15	10101	movBA	BX に AX の値を代入 BX = AX
0x16	10110	movii	アドレス BX にある命令語をアドレス AX に移動
Instruction Pointer Manipulation: 5			
0x17	10111	jmpo	ip に相補的テンプレートのアドレスを探索しセット。両方向探索。
0x18	11000	jmpb	ip に相補的テンプレートのアドレスを探索しセット。後方探索。
0x19	11001	call	サブルーチンの呼びだし。ip の値をスタックに入れる。
0x1A	11010	ret	サブルーチンから呼びだし元に戻る。スタックの値を取りだし、その値のアドレスにジャンプする。
Biological and Sensory: 11			
0x1B	11011	adro	アドレス AX にもっとも近い相補的テンプレートを探す。両方向探索。
0x1C	11100	adrb	アドレス AX の後方でもっとも近い相補的テンプレートを探す。後方探索。
0x1D	11101	adrf	アドレス AX の前方でもっとも近い相補的テンプレートを探す。前方探索。
0x1E	11110	mal	娘細胞（デジタル生物）用のメモリ領域を確保する
0x1F	11111	divide	娘細胞の分裂
Total: 32			

Table 2.2: 命令セット 0

2.7.2 命令セット0の先祖種0080aaa

以下に先祖種0080aaaのコードを示す。なお、この表では先祖種がメモリアドレス30000にあるものと仮定して、紙上シミュレーションを行った。

Addr.	命令語	説明	next ip	AX	BX	CX	DX
自己検査コードブロック							
30000	nop1	コード開始アドレステンプレート	30001	-	-	-	-
30001	nop1		30002	-	-	-	-
30002	nop1		30003	-	-	-	-
30003	nop1		30004	-	-	-	-
30004	zero	CXレジスタに値0を格納。	30005	-	-	0	-
30005	not0	CXレジスタの第1ビットに1をセット	30006	-	-	1	-
30006	shl	CXレジスタの値を1ビット左シフト	30007	-	-	2	-
30007	shl	CXレジスタの値をさらに1ビット左シフト	30008	-	-	4	-
30008	movDC	DX=CX, DXにテンプレートサイズが格納される	30009	-	-	4	4
30009	adrb	後続の一連のnop命令によるテンプレートと相補的なテンプレートのアドレス(相補的なテンプレートの最後のnop命令の次のアドレス)を後方(アドレス値が減る方向に)探索し、その値はAXレジスタに格納される。	30014	30004	-	4	4
30010	nop0	コード開始アドレステンプレートと相補的なテンプレート	-	-	-	-	-
30011	nop0		-	-	-	-	-
30012	nop0		-	-	-	-	-
30013	nop0		-	-	-	-	-
30014	subAAC	AX = AX - CXを計算。これによりコード開始アドレステンプレートの最初のアドレス、すなわち開始アドレスが求まる。その値はAXレジスタに格納される。	30015	30000	-	4	4
30015	movBA	BX = AXを行う。すなわち母セルのコード開始アドレスの値をBXレジスタにコピーする。	30016	30000	30000	4	4
30016	adrf	後続の一連のnop命令によるテンプレートと相補的なテンプレートのアドレスを前方(アドレス値が増える方向)に探索し、その値はAXレジスタに格納される。	30021	30079	30000	4	4
30017	nop0	コード終了アドレステンプレートと相補的なテンプレート	-	30079	30000	4	4
30018	nop0		-	30004	-	4	4
30019	nop0		-	30079	30000	4	4
30020	nop1		-	30079	30000	4	4
30021	incA	AX = AX + 1を計算。このゲノムコードにはコード終了アドレステンプレートの次に隣のゲノムとの区切りのためのダミー命令が1語含まれている(30079番地)。また、このダミー命令のアドレスが母セルのコード終了アドレスであるが、コードのサイズを正しく計算するためにAXレジスタの値に1を加えておく。	30022	30080	30000	4	4
30022	subCAB	CX = AX - BXを計算。コードサイズを計算し、CXレジスタに格納する。	30023	30080	30000	80	4

Table 2.3: 命令セット0の先祖種0080aaaの遺伝コード（その1）

Addr.	命令語	説明	next ip	AX	BX	CX	DX
自己複製ループのコードブロック							
30023	nop1	自己複製ループのアドレステンプレート	30024	30080	30000	80	4
30024	nop1		30025	30080	30000	80	4
30025	nop0		30026	30080	30000	80	4
30026	nop1		30027	30080	30000	80	4
30027	mal	娘セルの領域、この場合連続した 80 命令分の領域をスープ中に確保し、その先頭アドレスを AX レジスタに格納する。例として 31000 とする。	30028	31000	30000	80	4
30028	call	後続する一連の nop 命令によるテンプレートと相補的なテンプレートを探索し、そのテンプレートのアドレスにある命令の実行を行う。ここではコピープロシッジャーを呼び出している。	30044	31000	30000	80	4
30029	nop0	コピープロシッジャーの開始アドレステンプレートと相補的なテンプレート	-	31000	30000	80	4
30030	nop0		-	31000	30000	80	4
30031	nop1		-	31080	30000	80	4
30032	nop1		-	31000	30000	80	4
30033	divide	娘セルを独立させる。母セルは娘セルの領域への命令語の書き込みができなくなる。娘セルには命令ポインターが与えられ、スライス・キューとリーパー・キューに入る。	30034	31000	30000	80	4
30034	jmpo	後続する一連の nop 命令によるテンプレートと相補的なテンプレートの次の命令 (30027 mal) にジャンプする。これにより新たな娘セルの複製を始める。	30027	31000	30000	80	4
30035	nop0	自己複製ループのアドレステンプレートと相補的なテンプレート	-	31000	30000	80	4
30036	nop0		-	31000	30000	80	4
30037	nop1		-	31080	30000	80	4
30038	nop0		-	31000	30000	80	4
30039	ifz	コピープロシッジャーのテンプレートと区切るためのダミー命令。	-	31000	30000	80	4
コピー・プロシッジャー・コードブロック							
30040	nop1	コピープロシッジャーのテンプレート	-	31000	30000	80	4
30041	nop1		-	31000	30000	80	4
30042	nop0		-	31000	30000	80	4
30043	nop0		-	31000	30000	80	4
30044	pushA	AX レジスタの値をスタックに格納。娘セルのコード開始アドレスをスタックに退避しておく。	30045	31000	30000	80	4
30045	pushB	BX レジスタの値をスタックに格納。母セルのコード開始アドレスをスタックに退避しておく。	30046	31000	30000	80	4
30046	pushC	CX レジスタの値をスタックに格納。母セルのコードサイズをスタックに退避しておく。	30047	31000	30000	80	4
30047	nop1	コピーループ開始テンプレート	30048	31000	30000	80	4
30048	nop0		30049	31000	30000	80	4
30049	nop1		30050	31000	30000	80	4
30050	nop0		30051	31000	30000	80	4
30051	movii	BX レジスタに格納されている値のアドレスにある命令語を、AX レジスタに格納されている値のアドレスにコピーする。例えば、母セルのコード開始アドレス 30000 にある命令語 nop1 を娘セルのコード開始アドレス 31000 にコピーする。	30052	31000	30000	80	4

Table 2.4: 命令セット 0 の先祖種 0080aaa の遺伝コード（その 2）

Addr.	命令語	説明	next ip	AX	BX	CX	DX
コピー・プロシッジャー・コードブロック(続き)							
30052	decC	CXレジスタの値を1減ずる。たとえば80が79になる。	30053	31000	30000	79	4
30053	ifz	もしCXレジスタの値が0、すなわち母セルのすべての命令語を娘セルにコピーしおえたなら次の命令(30054 jmpo)を実行する。そうでないなら、それをスキップしてもう一つ後の命令(30055 nop0)を実行する。つまり、CXレジスタの値をだけコピーループが実行される。	30055	31000	30000	79	4
30054	jmpo	後続する一連のnop命令によるテンプレートと相補的なテンプレートの次の命令(30071 popC)にジャンプする。	30071	31000	30000	79	4
30055	nop0	コピープロシッジャー終了テンプレートと相補的なテンプレート	30056	31000	30000	79	4
30056	nop1		30057	31000	30000	80	4
30057	nop0		30058	31001	30001	80	4
30058	nop0		30059	31000	30000	79	4
30059	incA	AXレジスタの値を1増加する。母セルの次の命令語を指す。	30060	31001	30000	79	4
30060	incB	BXレジスタの値を1増加する。娘セルの次のアドレス値になる。	30061	31001	30001	79	4
30061	jmpo	後続する一連のnop命令によるテンプレートと相補的なテンプレートの次のアドレスにある命令(30051 movii)にジャンプする。次の命令語のコピーを行う。	30051	31001	30001	79	4
30062	nop0	コピーループ開始アドレステンプレートと相補的なテンプレート	-	31001	30001	79	4
30063	nop1		-	31001	30001	79	4
30064	nop0		-	31001	30001	79	4
30065	nop1		-	31001	30001	79	4
30066	ifz	テンプレートとテンプレートを区切るダミー命令	-	31001	30001	79	4
30067	nop1	コピープロシッジャー終了テンプレート	-	31001	30001	79	4
30068	nop0		-	31001	30001	80	4
30069	nop1		-	31001	30001	79	4
30070	nop1		-	31001	30001	79	4
30071	popC	スタックからCXレジスタに値(母セルのコードサイズ)を復帰させる。	30072	31001	30001	80	4
30072	popB	スタックからBXレジスタに値(母セルのコード開始アドレス)を復帰させる。	30073	31001	30000	80	4
30073	popA	スタックからAXレジスタに値(娘セルのコード開始アドレス)を復帰させる。	30074	31000	30000	80	4
30074	ret	コピープロシッジャーを終了して、このルーチンをcallした命令語の次の命令語を実行する。	30033	31000	30000	80	4
コード終了ブロック							
30075	nop1	コード終了アドレステンプレート	-	-	-	-	-
30076	nop1		-	-	-	-	-
30077	nop1		-	-	-	-	-
30078	nop0		-	-	-	-	-
30079	ifz	他のセルとの区切りのためのダミー命令	-	-	-	-	-

Table 2.5: 命令セット0の先祖種0080aaaの遺伝コード（その3）

2.7.3 パラサイト0045aaa

以下にパラサイト0045aaaのコードを示す。なお、この表では先祖種がメモリアドレス30000にあり、このパラサイトがメモリアドレス30080にあるものと仮定して、紙上シミュレーションを行った。

Addr.	命令語	説明	next ip	AX	BX	CX	DX
自己検査コードブロック							
30080	nop1	コード開始アドレステンプレート	30081	-	-	-	-
30081	nop1		30082	-	-	-	-
30082	nop1		30083	-	-	-	-
30083	nop1		30084	-	-	-	-
30084	zero	CXレジスタに値0を格納。	30085	-	-	0	-
30085	not0	CXレジスタの第1ビットに1をセット	30086	-	-	1	-
30086	shl	CXレジスタの値を1ビット左シフト	30087	-	-	2	-
30087	shl	CXレジスタの値をさらに1ビット左シフト。	30088	-	-	4	-
30088	movDC	DX=CX, DXにテンプレートサイズが格納される。	30089	-	-	4	4
30089	adrb	後続の一連のnop命令によるテンプレートと相補的なテンプレートのアドレス(相補的なテンプレートの最後のnop命令の次のアドレス)を後方探索し、その値をAXレジスタに格納する。	30090	30084	-	4	4
30090	nop0	コード開始アドレステンプレートと相補的なテンプレート	-	-	-	-	-
30091	nop0		-	-	-	-	-
30092	nop0		-	-	-	-	-
30093	nop0		-	-	-	-	-
30094	subAAC	AX = AX - CXを計算。これによりコード開始アドレステンプレートの最初のアドレスを求め、AXレジスタに格納する。	30095	30080	-	4	4
30095	movBA	BX = AXを行う。	30096	30080	30080	4	4
30096	adrf	後続の一連のnop命令によるテンプレートと相補的なテンプレートのアドレスを前方探索し、その値をAXレジスタに格納する。	30097	30124	30080	4	4
30097	nop0	コード終了アドレステンプレートと相補的なテンプレート	-	-	-	-	-
30098	nop0		-	-	-	-	-
30099	nop0		-	-	-	-	-
30100	nop1		-	-	-	-	-
30101	incA	AX = AX + 1を計算。このゲノムコードにはコード終了アドレステンプレートの次に隣のゲノムとの区切りのためのダミー命令が1語含まれている(30124番地)。また、このダミー命令のアドレスが母セルのコード終了アドレスであるが、次にコードのサイズを正しく計算するためにAXレジスタの値に1を加えておく。	30102	30125	30080	4	4
30102	subCAB	CX = AX - BXを計算。	30103	30125	30080	45	4

Table 2.6: パラサイト0045aaaの遺伝コード（その1）

Addr.	命令語	説明	next ip	AX	BX	CX	DX
自己複製ループのコードブロック							
30103	nop1	自己複製ループのアドレステンプレート	30104	30125	30080	45	4
30104	nop1		30105	30125	30080	45	4
30105	nop0		30106	30125	30080	45	4
30106	nop1		30107	30125	30080	45	4
30107	mal	娘セルの領域、この場合連続した45命令分の領域をスープ中に確保し、その先頭アドレスをAXレジスタに格納する。例として32000とする。	30108	32000	30080	45	4
30108	call	後続する一連のnop命令によるテンプレートと相補的なテンプレートを探索し、そのテンプレートのアドレスにある命令の実行を行う。ここでは先祖種のコピープロシッジャーを呼び出している。	30044	32000	30080	45	4
30109	nop0	コピープロシッジャーの開始アドレステンプレートと相補的なテンプレート	-	32000	30080	45	4
30110	nop0		-	32000	30080	45	4
30111	nop1		-	32000	30080	45	4
30112	nop1		-	32000	30080	45	4
30113	divide	娘セルを独立させる。母セルは娘セルの領域への命令語の書き込みができなくなる。娘セルには命令ポインターが与えられ、スライス・キューとリーバー・キューに入る。	30114	32000	30080	45	4
30114	jmpo	後続する一連のnop命令によるテンプレートと相補的なテンプレートの次の命令(30107 mal)にジャンプする。これにより新たな娘セルの複製を始める。	30107	32000	30080	45	4
30115	nop0	自己複製ループのアドレステンプレートと相補的なテンプレート	-	-	-	-	-
30116	nop0		-	-	-	-	-
30117	nop1		-	-	-	-	-
30118	nop0		-	-	-	-	-
30119	ifz	コピー・プロシッジャーのテンプレートと区切るためのダミー命令。	-	-	-	-	-
コード終了ブロック							
30120	nop1	コピー・プロシッジャーの開始テンプレートがコード終了テンプレートに変異	-	-	-	-	-
30121	nop1		-	-	-	-	-
30122	nop1		-	-	-	-	-
30123	nop0		-	-	-	-	-
30124	pushA	他のセルとの区切りのためのダミー命令	-	-	-	-	-

Table 2.7: パラサイト0045aaaの遺伝コード（その2）

2.7.4 ハイパー・パラサイト0078aaa

ハイパー・パラサイト0078aaaのコードを示す。なお、この表ではこのパラサイトがメモリアドレス30080にあり、このハイパー・パラサイトがメモリアドレス30125にあるものと仮定して、紙上シミュレーションを行った。

Addr.	命令語	説明	next ip	AX	BX	CX	DX
自己検査コードブロック							
30125	nop1	コード開始アドレステンプレート	30126	-	-	-	-
30126	nop1		30127	-	-	-	-
30127	nop0		30128	-	-	-	-
30128	adrb	コード開始アドレスを求める。テンプレートサイズをCXに格納する。	30132	30128	-	3	-
30129	nop0	コード開始アドレスと相補的なテンプレート	-	-	-	-	-
30130	nop0						
30131	nop1						
30132	subAAC	AX=AX-CXによりコード開始テンプレートの値を計算。	30133	30125	-	3	-
30133	movBA	BX=AX	30134	30125	30125	3	-
30134	adrf	コード終了テンプレート探索	30138	30203	30125	3	-
30135	nop0	コード終了テンプレートと相補的なテンプレート	-	-	-	-	-
30136	nop0						
30137	nop0						
30138	jmpb	突然変異でできた無意味な命令	30139	30203	30125	3	-
30139	zero	CX=0	30140	30203	30125	0	-
30140	subCAB	CX = AX - BX	30141	30203	30125	78	-
自己複製ループのコードブロック							
30141	mal	娘セルの領域を確保。その領域の先頭アドレスをAXに格納する。例としてAX=31000とする。	30142	31000	30125	78	-
30142	call	コピー・プロシッジャーの呼びだし。	30156	31000	30125	78	-
30143	nop0	コピー・プロシッジャーと相補的なテンプレー	-	-	-	-	-
30144	nop0						
30145	nop1						
30146	nop1						
30147	divide	娘セルの独立。	30148	31000	30125	78	-
30148	jmpb	コード開始テンプレートにジャンプ	30128	31000	30125	78	-
30149	nop0	コード開始テンプレートと相補的なテンプレート	-	-	-	-	-
30150	nop0						
30151	nop1						
30152	jmpo	突然変異でできた無意味なコード	-	-	-	-	-
30153	nop1						
30154	nop0						
30155	ifz	コピー・プロシッジャーとの区切り					

Table 2.8: ハイパー・パラサイト0078aaaの遺伝コード（その1）

Addr.	命令語	説明	next ip	AX	BX	CX	DX
コピー・プロシッジャーのコードブロック							
30156	nop1	コピープロシッジャー開始テンプレート	-	-	-	-	-
30157	nop1						
30158	nop0						
30159	nop0						
30160	pushax	レジスタの値をスタックに退避する。	30161	31000	30125	78	-
30161	pushbx		30162	31000	30125	78	-
30162	pushcx		30163	31000	30125	78	-
30163	nop1	コピー・ループ開始テンプレート	30164	31000	30125	78	-
30164	nop0		30165	31000	30125	78	-
30165	nop1		30166	31000	30125	78	-
30166	nop0		30167	31000	30125	78	-
30167	movii	命令語のコピー	30168	31000	30125	78	-
30168	decC	CX = CX-1	30169	31000	30125	77	-
30169	ifz	if(CX==0) なら次行実行、そうでないなら次次行を実行。	30170	31000	30125	77	-
30170	jmpo	自己複製ループの divide 命令にジャンプする。	30147	31077	30202	0	-
30171	nop1	コピー・プロシッジャー呼びだしテンプレートと相補的なテンプレート	30172	31000	30125	77	-
30172	nop1		30173	31000	30125	77	-
30173	nop0		30174	31000	30125	77	-
30174	nop0		30175	31000	30125	77	-
30175	incA	AX = AX + 1	30176	31001	30125	77	-
30176	incB	BX = BX + 1	30176	31001	30126	77	-
30177	jmpb	movii にジャンプ	30167	31001	30126	77	-
30178	nop0	コピーループ開始テンプレートと相補的なテンプレート	-	-	-	-	-
30179	nop1						
30180	nop0						
30181	nop1						
30182	jmpb	以下は無意味なコード	-	-	-	-	-
30183	nop1						
30184	nop0						
30185	popbx						
30186	nop1						
30187	popcx						
30188	popbx						
30189	popbx						
30190	ret						
30191	ifz						
30192	ifz						
30193	ifz						
30194	ifz						
30195	ifz						
30196	ifz						
30197	ifz						
30198	ifz						
30199	nop0	コード終了テンプレート	-	-	-	-	-
30200	nop1						
30201	nop1						
30202	nop1						

Table 2.9: ハイパー・パラサイト 0078aaa の遺伝コード（その 2）

2.7.5 社会的ハイパー・パラサイト 0061acg

社会的ハイパー・パラサイト 0061acg のコードを示す。なお、この表ではこの社会的ハイパー・パラサイトがメモリアドレス 30125 にあるものと仮定し、さらにもう一つの社会的ハイパー・パラサイトが隣接しているものとして紙上シミュレーションを行った。

Addr.	命令語	説明	next ip	AX	BX	CX	DX
自己検査コードブロック							
30125	nop1	母セルコード開始アドレステンプレート	30126	-	-	-	-
30126	nop1		30127	-	-	-	-
30127	nop0		30128	-	-	-	-
30128	adrb	コード開始アドレスを求める。テンプレートサイズを CX に格納する。	30132	30128	-	3	-
30129	nop0	コード開始アドレスと相補的なテンプレート	-	-	-	-	-
30130	nop0						
30131	nop1						
30132	subAAC	AX=AX-CX によりコード開始テンプレートの値を計算。	30133	30125	-	3	-
30133	movBA	BX=AX	30134	30125	30125	3	-
30134	adrf	コード終了テンプレート探索	30138	30186	30125	3	-
30135	nop0	コード終了テンプレートと相補的なテンプレート	-	-	-	-	-
30136	nop0						
30137	nop0						
30138	jmpb	突然変異でできた無意味な命令	30139	30186	30125	3	-
30139	zero	CX=0	30140	30186	30125	0	-
30140	subCAB	CX = AX - BX	30141	30186	30125	61	-
自己複製ループのコードブロック							
30141	mal	娘セルの領域を確保。その領域の先頭アドレスを AX に格納する。	30142	31000	30125	61	-
30142	call	コピー・プロシッジャーの呼びだし。	30159	31000	30125	61	-
30143	nop0	コピー・プロシッジャーと相補的なテンプレート	-	-	-	-	-
30144	nop0						
30145	nop1						
30146	incB	無意味なコード					
30147	divide	娘セルの独立。	30148	31000	30125	61	-
30148	jmpb	別のソーシャル・ハイパー・パラサイトのテンプレートにジャンプ	30120	31000	30125	61	-
30149	nop0	別のソーシャル・ハイパー・パラサイトのテンプレートを使って、自己検査コードへたどり着く。	-	-	-	-	-
30150	nop1						
30151	nop0						
30152	call	無意味なコード					
30153	movAB	無意味なコード					
30154	pushcx	無意味なコード					
30155	ifz	コピープロシッジャーとの区切り					

Table 2.10: 社会的ハイパー・パラサイト 0061acg のコード（その 1）

Addr.	命令語	説明	next ip	AX	BX	CX	DX
コピープロシッジャーのコードブロック							
30156	nop1	コピープロシッジャー開始テンプレート	-	-	-	-	-
30157	nop1						
30158	nop0						
30159	nop0						
30160	pushax	レジスタの値をスタックに退避する。	30161	31000	30125	61	-
30161	pushbx		30162	31000	30125	61	-
30162	pushcx		30163	31000	30125	61	-
30163	nop1	コピーループ開始テンプレート	30164	31000	30125	61	-
30164	nop0		30165	31000	30125	61	-
30165	nop1		30166	31000	30125	61	-
30166	nop0		30167	31000	30125	61	-
30167	movii	命令語のコピー	30168	31000	30125	61	-
30168	decC	CX = CX-1	30169	31000	30125	60	-
30169	ifz	if(CX==0) なら次行実行、そうでないなら次の次の命令を実行。	30170	31000	30125	60	-
30170	jmpb	自己複製ループの divide 命令にジャンプする。	30147	31060	30185	0	-
30171	nop1	コピープロシッジャー呼びだしテンプレートと相補的なテンプレート	30172	31000	30125	60	-
30172	nop1		30173	31000	30125	60	-
30173	nop0		30174	31000	30125	60	-
30174	incA	AX = AX + 1	30175	31001	30125	60	-
30175	incB	BX = BX + 1	30176	31001	30126	60	-
30176	jmpb	movii にジャンプ	30167	31001	30126	60	-
30177	nop0	コピーループ開始テンプレートと相補的なテンプレート	-	-	-	-	-
30178	nop1						
30179	nop0						
30180	nop1						
30181	pushdx	無意味なコード	-	-	-	-	-
30182	popbx	無意味なコード	-	-	-	-	-
30183	nop1	コード終了テンプレート	-	-	-	-	-
30184	nop1						
30185	nop1						

Table 2.11: 社会的ハイパー・パラサイト 0061acg のコード（その 2）

2.7.6 チーター 0027aab

チーター 0027aab のコードを示す。なお、この表では社会的ハイパー・パラサイトがメモリアドレス 30125 にあるものと仮定し、さらこのチーターがメモリアドレス 30186 にあるものとして紙上シミュレーションを行った。

Addr.	命令語	説明	next ip	AX	BX	CX	DX
自己検査コードブロック							
30186	nop1	コード開始アドレステンプレート	30187	-	-	-	-
30187	nop1		30188	-	-	-	-
30188	nop0		30189	-	-	-	-
30189	adrb	コード開始アドレスを求める。テンプレートサイズを CX に格納する。	30193	30189	-	3	-
30190	nop0	コード開始アドレスと相補的なテンプレート	-	-	-	-	-
30191	nop0						
30192	nop1						
30193	subAAC	AX=AX-CX によりコード開始テンプレートの値を計算。	30194	30186	-	3	-
30194	movBA	BX=AX	30195	30189	30186	3	-
30195	adrf	コード終了テンプレート探索	30199	30213	30186	3	-
30196	nop0	コード終了テンプレートと相補的なテンプレート	-	-	-	-	-
30197	nop0						
30198	nop0						
30199	zero	CX=0	30200	30213	30186	0	-
30200	subCAB	CX=AX-BA	30201	30213	30186	27	-
30201	subCAB	CX=AX-BA （突然変異のため、無意味にもう一回）	30202	30213	30186	27	-
自己複製ループのコードブロック							
30202	mal	娘セルの領域を確保。その領域の先頭アドレスを AX に格納する。例として AX=31000 とする。	30203	31000	30186	27	-
30203	call	コピー・プロシッジャーの呼びだし。	30246	31000	30186	27	-
30204	nop0	コピー・プロシッジャーと相補的なテンプレート	-	-	-	-	-
30205	nop0	(隣接する社会的ハイパー・パラサイト					
30206	nop1	のコピー・プロシッジャーをコール。) 以下、社会的					
30207	nop1	ハイパー・パラサイトに自分を複製させる。					
30208	divide		-	-	-	-	-
30209	jmpb		-	-	-	-	-
30210	nop1	下の社会的ハイパー・パラサイトのテンプレートと融合。	-	-	-	-	-
30211	nop1						
30212	nop1	自己検査コードへ戻る。					
社会的ハイパー・パラサイトの自己検査コードブロック							
30213	nop1	コード開始アドレステンプレート	-	-	-	-	-
30214	nop1		-	-	-	-	-
30215	nop0		-	-	-	-	-

Table 2.12: チーター 0027aab のコード

2.7.7 高度に進化したゲノム 0072etq

高度に進化したゲノム 0072etq のコードを示す。なおここでは、このゲノムがメモリアドレス 0 番地にあるものとして、紙上シミュレーションを行った。

Addr.	命令語	説明	next ip	AX	BX	CX	DX
自己検査コードブロック							
00000	nop1	コード開始アドレステンプレート	00001	-	-	-	-
00001	adrb	コード開始アドレスを求める。テンプレートサイズを CX に格納する。	00003	00001	-	1	-
00002	nop0	コード開始アドレステンプレートと相補的なテンプレート	-	-	-	-	-
00003	divide	娘セルの独立。一回目は無効。	00004	00001	-	1	-
00004	subAAC	AX = AX - CX	00005	00000	-	1	-
00005	movab	BX = AX	00006	00000	00000	1	-
00006	adrf	相補的テンプレートのアドレスを求める。	00008	00035	00000	1	-
00007	nop0	相補的なテンプレート	-	-	-	-	-
00008	incA	AX = AX + 1 (サイズの半分の値)	00009	00036	00000	1	-
00009	call	無意味な命令	00010	00036	00000	1	-
00010	subCAB	CX = AX - BA	00011	00036	00000	36	-
00011	pushB	BX の値をスタックに入れる。00030 の ret 命令で先頭アドレスに戻れるようにするため。	00012	00036	00000	36	-
00012	shl	CX = CX × 2	00013	00036	00000	72	-
コピーループのコードブロック。一度に 3 命令づつコピーする。							
00013	mal	娘セルの領域確保 (01000 に確保したと仮定する。)	00014	01000	00000	72	-
00014	nop0	コピーループの開始テンプレート	00015	01000	00000	72	-
00015	movii	コードのコピー (1 回目)	00016	01000	00000	72	-
00016	decC	CX = CX -1	00017	01000	00000	71	-
00017	decC	CX = CX -1	00018	01000	00000	70	-
00018	jmpb	無意味な命令	00019	01000	00000	70	-
00019	decC	CX = CX -1	00020	01000	00000	69	-
00020	incA	AX = AX +1	00021	01001	00000	69	-
00021	incB	BX = BX +1	00022	01001	00001	69	-
00022	movii	コードのコピー (2 回目)	00023	01001	00001	69	-
00023	decC	CX = CX -1	00024	01001	00001	68	-
00024	incA	AX = AX +1	00025	01002	00001	68	-
00025	incB	BX = BX +1	00026	01002	00002	68	-
00026	movii	コードのコピー (3 回目)	00027	01002	00002	68	-
00027	decC	CX = CX -1	00028	01002	00002	67	-
00028	not0	CX の最下位ビットを反転	00029	01002	00002	66	-
00029	ifz	CX==0 なら次の次の命令を実行。そうでないなら次の命令を実行。	00031	01002	00002	66	-
00030	ret	上の ifz が真のとき、先頭アドレスにリターンする。	00000	01036	00036	0	-

Table 2.13: ゲノム 0072etq のコード (その 1)

Addr.	命令語	説明	next ip	AX	BX	CX	DX
コピーループのコードブロック。（つづき）							
00031	incA	AX = AX +1	00032	01003	00002	66	-
00032	incB	BX = BX +1	00032	01003	00003	66	-
00033	jmpb	コピーループの開始テンプレートにジャンプ	00015	01003	00003	66	-
00034	nop1	コピーループの開始テンプレートと相補的なテンプレート	-	-	-	-	-
00035	ifz	無意味な命令	-	-	-	-	-
00036	nop1	コードサイズの半分を示すテンプレート	-	-	-	-	-
以下は実際には利用されないジャンクコード							
00037	adrb	領域が72確保されているうちの残り36命令。	-	-	-	-	-
00038	nop0	コピーループで作成されたのではなく、	-	-	-	-	-
00039	divide	確保した領域に残っていたコードの残骸	-	-	-	-	-
00040	subAAC	である。	-	-	-	-	-
00041	movab		-	-	-	-	-
00042	adrf		-	-	-	-	-
00043	nop0		-	-	-	-	-
00044	incA		-	-	-	-	-
00045	call		-	-	-	-	-
00046	subCAB		-	-	-	-	-
00047	pushB		-	-	-	-	-
00048	shl		-	-	-	-	-
00049	mal		-	-	-	-	-
00050	nop0		-	-	-	-	-
00051	movii		-	-	-	-	-
00052	decC		-	-	-	-	-
00053	decC		-	-	-	-	-
00054	jmpb		-	-	-	-	-
00055	decC		-	-	-	-	-
00056	incA		-	-	-	-	-
00057	incB		-	-	-	-	-
00058	movii		-	-	-	-	-
00059	decC		-	-	-	-	-
00060	incA		-	-	-	-	-
00061	incB		-	-	-	-	-
00062	movii		-	-	-	-	-
00063	decC		-	-	-	-	-
00064	not0		-	-	-	-	-
00065	ifz		-	-	-	-	-
00066	ret		-	-	-	-	-
00067	incA		-	-	-	-	-
00068	incB		-	-	-	-	-
00069	jmpb		-	-	-	-	-
00070	nop1		-	-	-	-	-
00071	ifz		-	-	-	-	-

Table 2.14: ゲノム 0072etq のコード（その 2）

2.7.8 最も短い自己複製生物 0022aak

もっとも短いサイズで自己複製できる生物のコードを以下に示す。なおここでは、このゲノムがメモリアドレス0番地にあるものとして、紙上シミュレーションを行った。

Addr.	命令語	説明	next ip	AX	BX	CX	DX
自己検査コードブロック							
00000	nop1	コード開始アドレステンプレート	00001	-	-	-	-
00001	adrb	コード開始アドレスを求める。テンプレートサイズをCXに格納する。	00003	00001	-	1	-
00002	nop0	コード開始アドレステンプレートと相補的なテンプレート	-	-	-	-	-
00003	divide	娘セルの独立。一回目は無効。	00004	00001	-	1	-
00004	subAAC	AX = AX - CX	00005	00000	-	1	-
00005	movab	BX = AX	00006	00000	00000	1	-
00006	adrf	相補的テンプレートのアドレスを求める。	00008	00021	00000	1	-
00007	nop0	コード終了テンプレートと相補的なテンプレート	-	-	-	-	-
00008	incA	AX = AX + 1 (コードの最後のダミー命令を含ませるため)	00009	00022	00000	1	-
00009	subCAB	CX = AX - BA (サイズ計算)	00010	00022	00000	22	-
00010	mal	娘セルの領域確保 (01000 に確保したと仮定する。)	00011	01000	00000	22	-
00011	pushB	BX の値をスタックに入れる。00017 の ret 命令で先頭アドレスに戻れるようにするため。	00012	01000	00000	22	-
コピーループのコードブロック。							
00012	nop0	コピーループの開始テンプレート	00013	01000	00000	22	-
00013	movii	コードのコピー	00014	01000	00000	22	-
00014	decC	CX = CX -1	00015	01000	00000	21	-
00015	ifz	CX==0 なら次の次の命令を実行。そうでないなら次の命令を実行。	00017	01000	00000	21	-
00016	ret	上の ifz が真のとき、先頭アドレスにリターンする。	00000	01021	00021	0	-
00017	incA	AX = AX +1	00018	01001	00000	21	-
00018	incB	BX = BX +1	00019	01001	00001	21	-
00019	jmpb	コピーループの開始に戻る	00013	01001	00001	21	-
00020	nop1	コピーループ開始テンプレートとの相補的テンプレート	-	-	-	-	-
00021	movii	コードの終了を表すダミーコード	-	-	-	-	-

Table 2.15: ゲノム 0022aak のコード

2.7.9 多細胞生物の先祖種 0082aaa

2.4節で紹介した多細胞生物の先祖種0082aaaのプログラム・コードを示す。

Addr.	命令語	説明
自己検査コードブロック		
30000	nop1	コード開始アドレステンプレート
30001	nop1	
30002	nop1	
30003	nop1	
30004	adrb	コード開始アドレスを探索し、AXにセット。DX=テンプレートサイズ
30005	nop0	コード開始アドレステンプレートの相補的テンプレート
30006	nop0	
30007	nop0	
30008	nop0	
30009	subAAC	AX=AX-CX
30010	movBA	BX=AX,BX=コード開始アドレス
30011	adrf	コード終了アドレスを探索し、AXにセット
30012	nop0	コード終了アドレステンプレートの相補的テンプレート
30013	nop0	
30014	nop0	
30015	nop1	
30016	incA	AX=AX+1（コード終了アドレスにダミー命令(30081)の分を加える）
30017	subCAB	CX=AX-BX（ゲノムサイズ）
自己複製ループのコードブロック		
30018	nop1	自己複製ループの開始テンプレート
30019	nop1	
30020	nop0	
30021	nop1	
30022	mal	娘セルのメモリを確保し、その先頭アドレスをAXにセット
30023	zeroD	DX=0
30024	zeroD	コードサイズを2の倍数にするためのダミー命令
30025	split	CPU分裂
30026	call	コピー・プロシッジャー呼出し
30027	nop0	コピー・プロシッジャーの開始テンプレートと相補的なテンプレート
30028	nop0	
30029	nop1	
30030	nop1	

Table 2.16: 多細胞生物の先祖種0082aaa（その1）

Addr.	命令語	説明
		自己複製ループのコードブロック（続き）
30031	join	すべてのCPUがこの命令語に到達するまで停止。CPUを一つにする。
30032	divide	娘セルの独立
30033	jmpo	自己複製ループの先頭にジャンプ
30034	nop0	自己複製ループの開始と相補的なテンプレート
30035	nop0	
30036	nop1	
30037	nop0	
30038	ifz	テンプレートの区切り
		コピー・プロシッジャーのコードブロック
30039	nop1	コピー・プロシッジャー・テンプレート
30040	nop1	
30041	nop0	
30042	nop0	
30043	pushA	AXの値（娘セルの先頭アドレス）をスタックに入れる
30044	pushB	BXの値（コード開始アドレス）をスタックに入れる
30045	pushC	CXの値（コードサイズ）をスタックに入れる
30046	shr	CXの値を右シフトCX=CX/2
30047	offAACD	AX=AX+CX × DX, 各CPU毎にコピー先のアドレスを計算する
30048	offBBCD	BX=BX+CX × DX, 各CPU毎にコピー元のアドレスを計算する
30049	nop1	コピーループ開始テンプレート
30050	nop0	
30051	nop1	
30052	nop0	
30053	movii	命令語をアドレスBXからアドレスAXへコピー
30054	decC	CX=CX-1
30055	ifz	CX==0なら次行実行、そうでないならスキップ
30056	jmpo	コピープロシッジャー終了テンプレートにジャンプ
30057	nop0	コピープロシッジャー終了テンプレートと相補的テンプレート
30058	nop1	
30059	nop0	
30060	nop0	
30061	incA	AX=AX+1
30062	incB	BX=BX+1
30063	jmpb	コピーループ開始にジャンプバック
30064	nop0	コピーループ開始と相補的テンプレート
30065	nop1	
30066	nop0	
30067	nop1	
30068	ifz	テンプレートの区切り
30069	nop1	コピープロシッジャー終了テンプレート
30070	nop0	
30071	nop1	
30072	nop1	
30073	popC	スタックから値（コードサイズ）を取りだしCXにセット
30074	popB	スタックから値（コード開始アドレス）を取りだしBXにセット
30075	popA	スタックから値（娘セルの開始アドレス）を取りだしAXにセット
30076	ret	呼出し元に戻る
		コード終了ブロック
30077	nop1	コード終了テンプレート
30078	nop1	
30079	nop1	
30080	nop0	
30081	ifz	他の生物との区切り

Table 2.17: 多細胞生物の先祖種0082aaa（その2）

2.7.10 進化した多細胞生物 0040aba

2.4節で紹介したゲノム 0040aba のプログラム・コードを示す。

Addr.	命令語	説明
30000	nop0	コード開始テンプレート
30001	adrb	コード開始アドレスを探索し AX にセット、テンプレートサイズを CX にセット
30002	nop1	コード開始テンプレートと相補的なテンプレート
30003	subAC	AX=AX-CX,AX からテンプレートサイズを引いてコード開始アドレスが求まる
30004	movAB	BX=AX,BX= コード開始アドレス
30005	adrf	コード終了アドレスを探索し AX にセット
30006	nop0	コード終了アドレスと相補的なテンプレート
30007	nop0	
30008	subCAB	CX = AX-BX,CX= コードサイズ
30009	mal	娘セルのメモリを確保し、その先頭アドレスを AX に入れる
30010	incC	イントロン (since CX no longer used)
30011	split	CPU が 2 個になる
30012	ifz	イントロン (since CX [size + 1] couldn't be zero)
30013	movCD	イントロン (since previous ifz could never be true)
30014	shr	CX=CX/2,CX= コードサイズの半分
30015	offAACD	AX=AX+CX × DX, 各 CPU 毎にコピー先のアドレスを計算する
30016	offBBCD	BX=BX+CX × DX, 各 CPU 毎にコピー元のアドレスを計算する
30017	zeroD	DX=0,2 度目の split の前に DX を 0 にする
30018	pushB	BX（コード開始アドレス）の値をスタックに格納。後の ret 命令でコードの先頭に戻れるようにするため。
30019	shr	CX=CX/2 CX= コードサイズの 1/4
30020	split	CPU が 4 個になる
30021	offAACD	各 CPU のコピー先のアドレスを計算
30022	offBBCD	各 CPU のコピー元のアドレスを計算
30023	nop1	コピーループ開始テンプレート
30024	nop0	
30025	movii	命令語をコピー
30026	decC	CX=CX-1
30027	ifz	CX==0 なら次行実行、そうでないならスキップ
30028	jmp	前方の join 命令までジャンプ
30029	nop0	join 命令にジャンプするためのテンプレート
30030	incA	AX=AX+1,increment source address
30031	incB	BX=BX+1,increment destination address
30032	jmpb	コピーループの開始にジャンプ
30033	nop0	コピーループの開始と相補的なテンプレート
30034	nop1	
30035	join	すべての CPU がここにくるまで待機。CPU を 1 つする。
30036	divide	娘セルの独立
30037	ret	スタックの値（コード開始アドレス）を取りだし、そのアドレス戻る
30038	nop1	コード終了テンプレート
30039	nop1	

Table 2.18: ゲノム 0040aba

第 3 章

Tierra のインストールと実行

現在公開されている Tierra のバージョンは 5.0 である。ただし、これは Network Tierra には対応していない。ネットワークバージョンは現在 (1999 年 3 月) なお開発途中で、一般公開の予定はまだ未定である。なおこの章の内容は、文献 [Ray98b] を適宜邦訳したものである。

3.1 入手方法

Tierra Version5.0 は以下の ftp サイトから入手できる。

`ftp://alife.santafe.edu/pub/SOFTWARE/Tierra`

ここには Tierra のソースコードと実行形式、それにいくつかのドキュメントが置かれている。以下のファイルがある。

- pub/SOFTWARE/Tierra/README
- pub/SOFTWARE/Tierra/source.tar.gz（ソースコード。すべてのプラットフォームで共通。）
- pub/SOFTWARE/Tierra/tierra.html（本章の元になった文献。Tierra を使用する上で必要な情報が書かれている。）
- pub/SOFTWARE/Tierra/tierraAmiga.tar.gz（Amiga 用実行ファイル）
- pub/SOFTWARE/Tierra/tierraLinux.tar.gz（Linux 用実行ファイル）
- pub/SOFTWARE/Tierra/tierraWin95.tar.gz（Windows95(98) 用実行ファイル）
- pub/SOFTWARE/Tierra/tierraWinNT.tar.gz（WindowsNT 用実行ファイル）
- pub/SOFTWARE/Tierra/tierrdos.tar.gz（MS-DOS 用実行ファイル）

3.2 インストールと起動

3.2.1 実行形式ファイルをダウンロードする場合

Linux,Windows95,98,NT,MS-DOS,Amiga の場合は、ダウンロード用ファイルの中にすでに実行形式が含まれている。インストールの仕方は以下の通り。

1. あなたの PC の適当な場所に Tierra をインストールするためのディレクトリ（フォルダ）を作る。

Windows95等の場合、

（例） `C:\tierra`

Linuxの場合、

（例） `mkdir ./tierra`

2. それぞれのOSに対応するtierra.X.tar.gzファイルを今作成したディレクトリにダウンロードする。ftpコマンドでダウンロードする場合、READMEとtierra.html以外はbinaryモードでダウンロードすること。

3. 先ほどのファイルを解凍する。Linuxの場合は、

```
% gunzip -c tierraLinux.tar.gz | tar oxvf -
```

のようにgunzipとtarコマンドでファイルを解凍する。

Windowsの場合はWinZipを使用する。WinZipはwebをサーチすれば見つかるでしょう。

4. 起動
Windows95などの場合、DOS windowを開き、以下のように入力する。

```
¥C:tierra > tierra si0
```

LINUXなどの場合も、以下のようにコマンドラインから入力する。

```
% tierra1 si0
```

ここでsi0はスープファイルと言って、Tierra実行時のパラメータを設定しているファイルである。si0は命令セット0用のスープファイルである。

3.2.2 ソースファイルをダウンロードする場合

ソースファイルをダウンロードし、自分でコンパイルして実行形式を作成する方法もある。SunなどのUNIXマシンの場合はこちらの方法でインストールしていただきたい。

1. あなたのコンピュータの適当な場所にTierraをインストールするためのディレクトリ（フォルダ）を作る。

UNIX OSの場合、

（例） `mkdir ./tierra`

2. source.tar.gz ファイルを今作成したディレクトリにダウンロードする。ftp コマンドでダウンロードする場合、binary モードでダウンロードすること。

3. 先ほどのファイルを解凍する。

```
% gunzip -c source.tar.gz |  tar oxvf -
```

解凍すると以下のようなディレクトリ構成になっている。

(a) tierra/
Tierra のソースコードおよびコンパイルのための Makefile、実行に必要なパラメータ設定ファイル、先祖種のコードなどがある。

(b) Bgl-GUI_X11/,Bgl-UI_stdio/,BglcInt/,Bglcom/,Bglserv/
Tierra 観察用ツール "Beagle" のソースコードがある。本資料ではこの Beagle についての詳しい説明はいたしません。詳細およびお問い合わせは、Beagle 開発担当の吉川氏のホームページ (http://www.hip.atr.co.jp/~yosikawa/) をご覧ください。

(c) dosview/,tools/
DOS 版の古い Beagle のソースコード。上記の Beagle とは全く別物である。MS-DOS で VGA または CGA のグラフィックカードを使用した PC でなければ利用できない。

4. コンパイル
UNIX マシンでコンパイルするには C コンパイラーがなければならない。ほとんどの UNIX OS の場合 GNU C(gcc) でコンパイル可能。以下の Makefile から OS に合った Makefile を選択する。

- **Makefile**: SunOS4.1.x 用。
- **Makefile-Alpha**:DEC-Alpha(Digital UNIX) 用
- **Makefile-Amiga**:Amiga 用
- **Makefile-Linux**:Linux 用
- **Makefile-SGI**:Silicon Graphics(IRIX) 用
- **Makefile-Solaris**:Sun Solaris2.x 用

なお、これらの Makefile の他に MakeBgl,MakeBgl-Alpha,MakeBgl-Linux,MakeBgl-SGI, MakeBgl-Solaris というのがあが、これらは前項で説明した Beagle で観察できる Tierra をコンパイルするためのものである。Beagle を使用するためにはこれらの Makefile を使用する。詳しくは前項で紹介したホームページを参照のこと。またその他に MakeAlmond,MakeMon などもあるが、これらは古い Makefile で現在は使用していない。
コンパイルの仕方は例えば Linux の場合、コマンドラインから、

```
% make -f Makefile-Linux
```

とする。すると実行ファイル“tierra1”,先祖種のコードを機械語に翻訳するアセンブラ“arg1”ができる。

5. アセンブル
tierraを起動する前に先祖種のコードをアセンブルする。先祖種は命令セットの種類によって異なる。ジーンバンク・ディレクトリgb0には命令セット0の先祖種のコード0080aaa.tieがある。これは以下のようにアセンブルし、機械語コードファイル（ジーンバンクファイル）0080.genを作成する。機械語コードファイルの名前は、先祖種のコードサイズが80ならば、0080.genというように**“サイズ(4桁)”.gen**と命名する。

```
% cd gb0
% ../arg1 cv 0080.gen 0080aaa.tie
```

すると、

```
creating archive ‘‘0080.gen’’
a - 0080aaa 0.831 0.424 0
```

というメッセージとともに、アセンブルされる。

6. 起動
コマンドラインから以下のように入力すると、数秒後にTierraがスタートする。

```
% tierra1 si0
```

ここでsi0はスープファイルと言って、Tierra実行時のパラメータを設定しているファイルである。si0は命令セット0用のファイルである。このファイルの最終行に、前項で作成した先祖種（今の例では**0080aaa**）が記述されていることを確認しておくこと。

3.3 ジーンバンク (Genebank)

スープファイルのパラメーターGeneBnkerを1にセットすると、デジタル生物が生まれるごとにコードを親のコードと比較する。そして親のコードと異なる場合は、新しい名前（サイズ＋アルファベット3文字）が付与される。またそのコードを持った生物がパラメータ**SavMinNum,SavThr,SavThrPop**の値を越えた場合、そのコードはジーンバンクに保存される。ジーンバンク・ファイルはコードサイズ毎に作られる。例えば命令セット0でサイズ80なら**gb0**の下のファイル**0080.gen**に機械語形式で保存される。

3.4 画面メニュー

3.4.1 基本画面 (The basic Screen)

Tierraがスタートして少しすると以下のような画面が現れる。

```
InstExec  =  18036  Cells =   16
off

InstExeC          0  Generation       0  918539648     Tue Feb  9 14:54:08 1999
NumCells         16  Speed          200
RateMut       12800  RatMovMut        0  RateFlaw      51200  AgeAvg            0
AvgSize          80  AvgPop           0  FecundAvg       0.0  RepInstEf       0.0
Births            0
Deaths            0

                    Press Interrupt Key for Menu ...
```

またはスープファイルのパラメーター GeneBnker の値を 1 にセットして、起動しなおすと、以下のような画面が現れる。

```
InstExec  = 1,032288  Cells =  338  Genotypes =  123  Sizes =  60
Extracted = 0080aaa @ 241

InstExeC          1  Generation       2  918540532     Tue Feb  9 15:08:52 1999
NumCells        338  NumGen         123  NumSizes         60  Speed          7142
RateMut       19786  RatMovMut        0  RateFlaw      53376  AgeAvg         1877
AvgSize          82  AvgPop         271  FecundAvg       1.5  RepInstEf      10.5
Births          737
Deaths          417
MaxGenPop       193  (0080aaa) MaxGenMem      193 (0080aaa)  NumGenDG          1

                    Press Interrupt Key for Menu ...
```

この画面の上部 2 行は **STATS** エリアと呼び、特に大事な情報をデジタル生物が誕生する度に更新し表示する。その下の数行は **PLAN** エリアと呼び、Tierra の実行状況を示すいくつかの統計情報などを表示する。このエリアの情報は 100 万命令実行毎に更新される。最終行は **HELP** エリアと呼び、UNIX 環境では Ctrl キーと C で、DOS(Windows) 環境では Back Space キーで、次節で説明するメニューを表示する。
下の方の画面例を元に、表示内容を説明する。

(1)**STATS** エリアの情報

- **InstExec=1,032288** は、シミュレーション開始から 1,032,288 回の命令が実行された事を表示。
- **Cells = 338** は、スープメモリーの中に 338 個体の成人した cell が生息していることを表示。
- **Genotypes = 123** は、このスープメモリーの中に 123 種類のゲノタイプ（種）がある事を表示。
- **Sizes = 60** は、このスープメモリーの中に 60 種類のサイズのゲノタイプ（種）がある事を表示。
- **Extracted = 0080aaa @ 241** は、ジーンバンクに登録されたゲノタイプの名前 (0080aaa) とその個体数 (241) を表示。

(2)**PLAN** エリアの情報

- **InstExeC = 1** は、このPLAN情報表示時点で、シミュレーション開始から100万回以上の命令が実行された事を表示。
- **Generation = 2** は、このPLANエリア情報表示時点で2世代目のデジタル生物たちが生息している事を表示。
- **918540532 Tue Feb 9 15:08:52 1999** は、このPLAN情報表示時点の時刻である。最初の数字はUNIXの32ビット整数値の形式で時刻を表示。
- **NumCells = 338** は、このPLANエリア情報表示時点でスープメモリ中に338個体のデジタル生物が生息している事を表示。
- **NumGen = 123** は、このPLANエリア情報表示時点で123種類のゲノムタイプ（種）が生息している事を表示。
- **NumSizes = 60** は、このPLANエリア情報表示時点で60種類のゲノムサイズが存在する事を示している。
- **Speed = 7142** は、この100万命令実行期間における命令実行スピードで、1秒間に7142回の命令が実行された事を示す。
- **RateMut = 19786** は、この100万命令実行期間における、バックグランド（放射線による）突然変異が19786/2命令に1回の割合で起こった事を示す。
- **RatMovMut = 0** は、この100万命令実行期間における、複製エラーによる突然変異が0/2回に1回の割合で起こった事を示す。
- **RateFlaw = 53376** は、この100万命令実行期間における、命令の実行が不適切に行われたのが53376/2回に1回の割合で起こった事を示す。
- **AgeAvg = 1877** は、この100万命令実行期間のデジタル生物の平均寿命（実行した命令回数）。
- **AvgSize = 82** は、この100万命令実行期間のデジタル生物の平均ゲノムサイズ。
- **AvgPop = 271** は、この100万命令実行期間のデジタル生物の平均人口。
- **FecundAvg = 1.5** は、この100万命令実行期間のデジタル生物の平均自己複製回数。
- **RepInstEf = 10.5** は、「**repinst/ゲノムサイズ**」のこの100万命令実行期間における平均。**repinst** は一回の自己複製に必要な命令の実行回数である。よってこれは自己複製の効率を表す。
- **Births = 737** は、この100万命令実行期間中において誕生したデジタル生物の数。
- **Deaths = 417** は、この100万命令実行期間中における死亡数。

- **MaxGenPop = 193 (0080aaa)** は、この時点で最も数が多いゲノムタイプの数とその名前を表示。
- **MaxGenMem = 193 (0080aaa)** は、この時点で最もスープメモリーをたくさん占有しているゲノムタイプの数と名前を表示。
- **NumGenDG = 1** は、ジーンバンクに恒久的に登録されたゲノムタイプが1種類である事を示す。

3.4.2 メインメニュー

最終行のメニューは以下のようになっている。これをメインメニューと呼ぶ。

```
i-info  v-var  s-save  S-shell  q-save&quit  Q-quit  m-misc  c-continue |->
```

- **i-info:** ジーンバンクに格納されている情報を表示させるメニュー。
- **v-var:** スープファイルに設定されているパラメータを表示したり、変更したりさせるメニュー。
- **s-save:** 実行途中で、システムの状態をファイルにセーブする。
- **S-shell:**Tierraを中断してDOSプロンプトまたはUNIXシェルを起動する。
- **q-save&quit:** この時点でシステムをセーブし、止める。
- **Q-quit:** この時点で止める。セーブはしない。
- **m-misc:** その他のメニューを表示する。
- **c-continue:** 実行続行する。

キーボードで"i"と入力すると、以下のようなメニュー(Infoメニュー)に切り替わる。

```
INFO | p-plan s-size g-gen m-mem z-size_query e-reprod_eff >
```

p-planメニューは先ほど紹介した**PLAN**メニューを再表示する。これは後述するヒストグラム表示を非表示にする。また他のメニューで表示が見苦しくなったときに、画面をクリアする事にも使用できる。

3.4.3 The Histograms

Infoメニューで、's','g','m','e'を入力すると、以下のようにメニューが変わる。'f'で頻度順にゲノムサイズの分布グラフが表示される。's'でサイズ順にゲノムサイズの分布グラフが表示される。

```
sort order | f-freq/size  s-size/freq ->
```

例えばs-sizeの場合で、"f"を選択すると、以下のように頻度順に各ゲノムサイズの個体数を表示する。この図で、1列目の数字はゲノムサイズを、2列目の数字はそのサイズのゲノムタイプ数を3列目は個体数を表す。

```
 80  31    216 | **************************************************
 79   4      9 | ***
 45   2      6 | **
103   4      5 | **
 83   1      4 | *
100   4      4 | *
160   4      4 | *
 52   3      3 | *
 78   1      3 | *
 96   1      3 | *
105   2      3 | *
113   2      3 | *
118   2      3 | *
```

g-genはゲノタイプ毎の個体数のヒストグラム、**m-mem**はゲノムサイズ毎のメモリー占有率のヒストグラム、**e-reprod_eff**は、自己複製効率のヒストグラム表示する。

3.4.4 The Size Class Information Display

Infoメニューで"z"を選択し、調べたいゲノムサイズを入力すると、そのゲノムサイズの統計情報が表示される。

```
Gene:    #  Mem Errs   Move  Bits
aaq      1    0    0      0  EX      TC      TP      MF      MT      MB
aar      2    0    0      0  EX      TC      TP      MF      MT      MB
aas      1    0    0      0  EX      TC      TP      MF      MT      MB
aat      1    0    1     80  EX      TC      TP      MF      MT      MB
aav      1    0    0      0  EX      TC      TP      MF      MT      MB
aaw      1    0    0     80  EX      TC      TP      MF      MT      MB
aay     13    1    0     80  EX      TC      TP      MF      MT      MB
aba      2    0    0     22  EX      TC      TP      MF      MT      MB
abb      1    0    0      0  EX      TC      TP      MF      MT      MB
abc      2    0    0     80  EX      TC      TP      MF      MT      MB
abd      1    0    0      0  EX      TC      TP      MF      MT      MB
U/D-page up/down  q-quit->
```

#は個体数、**Mem**はこのゲノタイプが占有するスープメモリー（パーセント）、**Errs**は命令実行エラーの回数、**Move**は娘セルへのコピーされた命令の数である。**Bits**は観察用の**Watch Bit**というもので、後の**3.5節 soup_in**パラメーターの所で説明する。

3.4.5 v-varメニュー

メインメニューから'v'を入力すると、以下のようなサブメニュー(**Var**メニュー)が表示される。

```
VAR |  a-alter variable  e-examine variable->
```

ここで、'a'を入力すると、スープファイルで設定されている各パラメーターを変更する事ができる。また'e'を入力すると、同ファイルで設定されたパラメーターの値を表示する。

3.4.6 m-misc メニュー

オプションメニューから'm'を入力すると、以下のようなサブメニュー (Misc メニュー) が表示される。

```
H-Histo Log  I-Inj Gene  M-Micro Tog      B-Brk Menu  m-more->
```

- **H-Histo Log:** ログファイル tierra.log へのヒストグラムデータの書き込みをするかしないか、切り替えする。
- **I-Inj Gene:** スープメモリーにゲノムを注入する事ができる。
- **M-Micro Tog:** 仮想デバッガーの実行モードを切り替える事ができる。仮想デバッガーには delay,keypress,off の 3 つのモードがある。delay モードは命令語に実行の様子を一秒毎に表示する。keypress モードでは、キーを押す毎にその実行の様子を表示する。off モードは、仮想デバッガーを off の状態にし、通常の実行状態にする。
- **B-brk Menu:** 仮想デバッガーの BreakPoint を設定する。

次項でこの仮想デバッガーについて説明する。

3.4.7 仮想デバッガー

もし、シミュレーションの開始からデバッガーを起動したい場合は、tierra 実効時に二つの引数を与えます。一つ目はいつも通りスープファイル名で、二つ目は適当な文字を与えます。すると最初の命令が実行される直前でメインメニューが表示される。その状態で Misc メニューを選択し、Micro Tog メニューで仮想デバッガーを keypress モードにする。

```
% tierra1 si0 junk
```

仮想デバッガーでは PLAN エリアの情報が以下のように変わる。

```
  Cell  1: 42 000075aaa @   38305 Slice=  39 Stack [      38338]
    IP [      38338] ( 33     ) = 0x1f    divide   [      45923]
OSD AX [      45923] ----------- host ------------ [      38305]
    BX [      38305]     id : 0075aaa              [         75]
    CX [         75] status : alive               [          0]
    DX [          4] offset :     33             [          0]
    EX [          0] inst   : 1756               [          0]
    FX [          0] instP  : 1756               [          0]
    Flag:  0   Stk:  9                           [          0]
    NO Daughter                                  [          0] <
             0 of   1 Cpus  thread [     0]
```

- **Cell 1: 42 000075aaa @ 38305 Slice= 39**
  それぞれのセル（デジタル生物）構造体は 2 位元の配列に格納されている。この例ではこのセルは配列の (1,42) にある事を示している。またこのセルのゲノム名は 0075aaa でありスープアドレス 38305 番地に存在している。**Slice= 39** は、このタイムスライスではあと 39 回、命令を実行できる事を示している。

- **IP [38338] (33) = 0x1f divide**
  IP(命令ポインター)アドレスが38338にあり、このセルの先頭から33個離れた命令語を実行している。その命令語の機械語コードは**0x1f**でニーモニックは**divide**である。

- **OSD AX [45923]**
  AXレジスタの値が45923である事を表している。以下BX～FXレジスタの値も表示している。**OSD**の各アルファベッドは'O':offset register, 'S':source register, 'D':destination registerとしてこのレジスタが使用されている事を示している。

- **——— host ———**
  このゲノムがホスト（宿主）であることを示している。

- **id : 0075aaa**
  はこのセルのゲノムタイプを表示している。

- **status : alive**
  このセルが生きている事を示している。

- **offset: 33**
  現在実行している命令語のこのセルの先頭からのオフセット位置である。

- **inst : 1756**
  このセルがこれまで実行した命令の回数。

- **instP : 1756**
  各スレッドで実行した命令の回数の合計。

- **Stack [38338]**
  以下はスタックの内容。

- **Flag: 0 Stk: 9**
  フラグレジスターの状態とスタックポインターの値。

- **No Daughter**
  現在娘セルがいない事を示す。

- **0 of 1 Cpus thread [0]**
  このセルにはcpuが一個あり、0番のcpu(スレッド）が実行中である事を示す。現在のTierraでは各セルは複数のcpuを持つ事が可能である。この例ではcpuは一個だが、複数の場合はどのcpuを実行しているかをこの部分で表示する。

Micro Togメニューでデバッガーをkeypressモードにすると、最下行が以下のようなサブメニューも変わる。

```
T-Trk cell  Trk U-cur N-New thd  r-Rpt Brk  q-Main d-displ a-disasm
```

または、

```
t-untrk cell  Trk u-cur r-Rpt Brk  q-Main d-displ a-disasm
```

など。それぞれ、'T' や 'U' などの '-' の前のアルファベットでメニューを選択する。

- **T-Trk cell:** 現在のセルの実行を追跡。(Track cell mode)
- **t-Untrk cell:**Track cell mode を解除。
- **Trk U-cur:** 現在のセルの cpu（スレッド）の実行を追跡。(Track cpu mode)
- **Trk u-cur:**Track cpu mode を解除。
- **N-New thd:** 現在のセルの最も新しい cpu（スレッド）を追跡。
- **r-Rpt Brk:** ブレークポイントの所まで実行を進める。
- **q-Main:** メインメニューに戻る。
- **d-displ:** スープメモリの状態を表示する。
- **a-disasm:** 指定したセルのコードを表示する。

3.5 soup_in パラメーター

典型的なスープファイルは以下のように記述されている。

```
# tierra core: 14-12-93 INST == 0

# observational parameters:

BrkupSiz = 1024      size of output file in K, named break.1, break.2 ...
CumGeneBnk = 0    Use cumulative gene files, or overwrite
debug = 0              0 = off, 1 = on, printf statements for debugging
DiskBank = 1    turn disk-genebanker on and off
DiskOut = 0           output data to disk (1 = on, 0 = off)
FindTimeM = 0    to set trap at a certain InstExe time, for debugging
FindTimeI = 0    to set trap at a certain InstExe time, for debugging
GeneBnker = 1   turn genebanker on and off
GenebankPath = gb0/  path for genebanker output
hangup = 0            0 = exit on error, 1 = hangup on error for debugging
MaxFreeBlocks = 800     initial number of structures for memory allocation
SaveFreq = 100     frequency of saving core_out, soup_out and list
SavRenewMem = 0   free and renew dynamic memory after saving to disk
SavMinNum = 10     minimum number of individuals to save genotype
SavThrMem = .02  threshold memory occupancy to save genotype
SavThrPop = .02  threshold population proportion to save genotype
TierraLog = 0          0 = no log file, 1 = write log file
WatchExe = 0     mark executed instructions in genome in genebank
WatchMov = 0     set mov bits in genome in genebank
WatchTem = 0     set template bits in genome in genebank

# environmental variables:

alive = 0       how many generations will we run, 0 = infinite
DistFreq = -.3     frequency of disturbance, factor of recovery time
DistProp = .2      proportion of population affected by distrubance
DivSameGen = 0 cells must produce offspring of same genotype, to stop evolution
DivSameSiz = 0 cells must produce offspring of same size, to stop size change
DropDead = 5 stop system if no reproduction in the last x million instructions
EjectRate = 0 rate at which random ejections from soup occur
GenPerBkgMut = 32 mutation rate control by generations ("cosmic ray")
```

```
GenPerFlaw = 32       flaw control by generations
GenPerMovMut = 0    mutation rate control by generations (copy mutation)
GenPerDivMut = 32
GenPerCroInsSamSiz = 32
GenPerInsIns = 32
GenPerDelIns = 32
GenPerCroIns = 32
GenPerDelSeg = 32
GenPerInsSeg = 32
GenPerCroSeg = 32
MutBitProp = .2   proportion of mutations that are bit flips
IMapFile = opcode.map map of opcodes to instructions, file in GenebankPath
JmpSouTra = 0.   source track switches per average size
JumpTrackProb = .2 probability of switching track during a jump of the IP
LazyTol = 10 tolerance for non-reproductive cells
MalMode = 1 0 = first fit, 1 = better fit, 2 = random preference,
# 3 = near mother's address, 4 = near bx address
# 5 = near top of stack address, 6 = suggested address (parse dependant)
MalReapTol = 1  0 = reap by queue, 1 = reap oldest creature within MalTol
MalSamSiz = 0  force memory alloc to be same size as parent (stop evolution)
MalTol = 20 multiple of avgsize to search for free block
MateSizeEp = 2 size epsilon for potential mate
MaxCpuPerCell = 16   maximum number of CPUs allowed per cell
MaxIOBufSiz = 8      maximum size for IOS buffer
MaxGetBufSiz = 4      maximum size for get IO buffer
MaxPutBufSiz = 4      maximum size for put IO buffer
MaxSigBufSiz = 8      maximum size for signal buffer
MemModeFree = 0  read, write, execute protection for free memory
MemModeMine = 0  rwx protection for memory owned by a creature
MemModeProt = 2  rwx protection for memory owned by another creature
# rwx protect mem: 1 bit = execute, 2 bit = write, 4 bit = read
MinCellSize = 12       minimum size for cells
MinGenMemSiz = 12       minimum size for genetic memory of cells
MinTemplSize = 1       minimum size for templates
MovPropThrDiv = .7       minimum proportion of daughter cell filled by mov
new_soup = 1          1 = this a new soup, 0 = restarting an old run
NumCells = 2      number of creatures and gaps used to inoculate new soup
PhotonPow = 1.5         power for photon match slice size
PhotonWidth = 8   amount by which photons slide to find best fit
PhotonWord = chlorophill  word used to define photon
PutLimit = 10 distance for intercellular communication, mult of avg creat siz
ReapRndProp = .3 top prop of reaper que to reap from
SearchLimit = 5 distance for template matching, mult of avg creat siz
seed = 0 seed for random number generator, 0 uses time to set seed
SizDepSlice = 0  set slice size by size of creature
SlicePow = 1    set power for slice size, use when SizDepSlice = 1
SliceSize = 25    slice size when SizDepSlice = 0
SliceStyle = 2   choose style of determining slice size
SlicFixFrac = 0   fixed fraction of slice size
SlicRanFrac = 2   random fraction of slice size
SoupSize = 60000 size of soup in instructions

MonPort = 17501
MigrCtrlPort = 17502
center
0080aaa
space 10000
0045aaa
```

スープファイルに設定されるパラメータ名は、1カラム目から書かれていなければならない。空白があると、正しいパラメータ名として認識されない。また1カラム目が#の行は、その行がコメントである事を示す。パラメータの順番は重要ではないが、先祖種のスープメモリ内の位置とゲノム名だけは最下行になければならない。

それぞれのパラメータの意味を以下に説明する。

3.5.1 # observational parameters: (観察用パラメータ)

observational parameter は、Tierraの実効条件自体には影響しない。実行中にどのような情報を得たいかにより、各パラメータを設定する。

- **BrkupSiz = 1024** size of output file in K, named break.1, break.2 ...
  この値が0の場合、デジタル生物の誕生と死の情報がtierra.runというファイルに記録される。0より大きな整数の場合、「この値×1Kbyte」毎にbreak.1,break.2...に記録される。つまりこの値が1024ということは、1Mbyteの大きさのファイルが作成されていく事を意味する。これらのファイルは後述する**GenebankPath**で設定されるディレクトリに作成される。

- **CumGeneBnk = 0** Use cumulative gene files, or overwrite
  この値が0の場合、新しいジーンバンクでスタートする。1の場合、既存のジーンバンクに上書きする。

- **debug = 0** 0 = off, 1 = on, printf statements for debugging
  Tierraの開発時に、デバッグ情報をプリントするためのフラグ。

- **DiskBank = 1** turn disk-genebanker on and off
  この値が1の場合、ゲノムをディスクに保存する。

- **DiskOut = 0** output data to disk (1 = on, 0 = off)
  この値が1の場合、デジタル生物の誕生と死の情報を前述したtierra.runまたはbreak.xファイルに保存する。

- **FindTimeM = 0** to set trap at a certain InstExe time, for debugging
  FindTimeI = 0 to set trap at a certain InstExe time, for debugging
  デバッグのために、ここでセットした命令実行回数でTierraの実行をストップさせる。

- **GeneBnker = 1** turn genebanker on and off
  この値が0のときジーンバンクの機能offに、それ以外の値のときonにする。ジーンバンクがoffのとき、デジタル生物の誕生と死およびそのサイズが記録されるが、ゲノムタイプは記録されない。またディスクへの保存もされない。onの場合、ゲノタイプはサイズとアルファベット3文字で与えられる。また後述する**SavThrMem,SavThrPop**の条件を越えたゲノムはジーン・バンクファイルに保存される。

- **GenebankPath = gb0/** path for genebanker output
  ジーンバンクのディレクトリ名。最後の文字はスラッシュ'/'でなければならない。

- **hangup = 0** 0 = exit on error, 1 = hangup on error for debugging
  この値が1の場合、もしTierraの実行に致命的なエラーがあっても終了せずに待機する。これはTierra開発時のデバッグで使用する。

- **MaxFreeBlocks = 800** initial number of structures for memory allocation
  空きメモリー探索時に使うメモリー構造体配列を作成するための初期値。

- **SaveFreq = 100** frequency of saving core_out, soup_out and list
  「この値×100万命令」実行毎に、すべての実行データを**core_out,soup_out**ファイルにセーブする。

- **SavRenewMem = 0** free and renew dynamic memory after saving to disk
  この値を1にセットすると、上の**SaveFreq**で実行データをセーブしたとき、確保されたメモリーを一旦全部解放し、ファイルからデータを読みだして改めてメモリーを確保する。これはメモリーリーク対策のために設けられた機構だが、まだ問題があるので0以外の値にはセットしないでください。

- **SavMinNum = 10** minimum number of individuals to save genotype
  この値より人口が多くなったゲノタイプに恒久的なゲノム名を与え、ジーンファイルに保存する。

- **SavThrMem = .02** threshold memory occupancy to save genotype
  SavThrPop = .02 threshold population proportion to save genotype
  それぞれ、あるゲノム（種）が全スープメモリ、全個体数に対して「この値×0.5」以上の割合になった場合に、恒久名を付与されてジーンバンクに保存される。0.5を乗じるのは親セルのみを考慮するため。

- **TierraLog = 0** 0 = no log file, 1 = write log file
  この値が0でない場合、Tierraの実行状態をモニターするログファイルtierra.logが作成される。

- **WatchExe = 0** mark executed instructions in genome in genebank
  ジーンバンク機能がonになっている場合、この値が1にセットされていると、それぞれの恒久ゲノタイプが実行している命令を観察できる。GList構造体(tierra.hで定義されている)には、そのセルが自分自身のコードを実行しているのか、他のセルのコードを実行しているのか、あるいは他のセルが自分のコードを実行しているのかを記録する以下のようなフィールドがある。

  bit 2 EXs = 自分自身のコードを実行 (self),
  bit 3 EXd = 娘セルのコードを実行、
  bit 4 EXo = 他のセルのコードを実行、(parasite)
  bit 5 EXf = 空きメモリーのコードを実行、
  bit 6 EXh = 他のセルに自分のコードが実行されている。(host)

- **WatchMov = 0** set mov bits in genome in genebank
  ジーンバンク機能がonになっている場合この値が1にセットされていると、どのセルがどのセルのコードを複製しているかを観察できる。

  bit 17 MFs = 自分自身のコードからコピー
  bit 18 MFd = 娘セルのコードからコピー
  bit 19 MFo = 他のセルのコードからコピー
  bit 20 MFf = 空きメモリーのコードからコピー
  bit 21 MFh = 他のセルが自分のコードからコピー (host)
  bit 22 MTs = 自分自身にコードをコピー
  bit 23 MTd = 娘セルにコードをコピー
  bit 24 MTo = 他のセルにコードをコピー
  bit 25 MTf = 空きメモリーにコードをコピー
  bit 26 MTh = 他のセルによってコードが書き込まれている (host)
  bit 27 MBs = 他のセルのコードを使って、自分のコードをコピーする
  bit 28 MBd = 他のセルのコードを使って、娘セルのコードをコピーする
  bit 29 MBo = 他のセルのコードを使って, 他のセルのコードをコピーする
  bit 30 MBf = 他のセルのコードを使って, 空きメモリーのコードをコピーする

bit 31 MBh = 他のセルが他のCPUを使って自分のコードをコピーする

- **WatchTem = 0** set template bits in genome in genebank
  ジーンバンク機能がonになっている場合この値が1にセットされていると、どのセルのテンプレートがどのセルのテンプレートとマッチしているかを観察する。

  bit 7 TCs = 自分自身のテンプレートと相補的にマッチしている
  bit 8 TCd = 娘セルのテンプレートと相補的にマッチしている
  bit 9 TCo = 他セルのテンプレートと相補的にマッチしている
  bit 10 TCf = 空きメモリーのテンプレートと相補的にマッチしている
  bit 11 TCh = 自分のテンプレートが他セルのテンプレートと相補的にマッチ(host)
  bit 12 TPs = 自分自身のテンプレートを使用
  bit 13 TPd = 娘セルのテンプレートを使用
  bit 14 TPo = 他セルのテンプレートを使用
  bit 15 TPf = 空きメモリーのテンプレートを使用
  bit 16 TPh = 他セルに自分のテンプレートが利用されている

3.5.2 # environmental variables: (環境設定パラメータ)

以下はTierraの実行環境に影響を与えるパラメータである。

- **alive = 0** how many generations will we run, 0 = infinite
  何世代までTierraを実行するかを設定する。0の場合は無限に実行を続ける。

- **DistFreq = -.3** frequency of disturbance, factor of recovery time
  "disturbance"(妨害)の発生頻度をコントロールする。disturbanceを起こさせたくない場合は負の値にセットする。この値が正だと、スープメモリーが一杯になってリーパーが起動し、**DistProp** × **SoupSize** の空きメモリーができるまでセルを殺し、再びスープメモリが一杯になるまでの時間を計測する。この時間(回復時間)をrtimeと呼ぶことにする。するとつぎのdisturbanceはスープメモリが一杯になってから(rtime × DistFreq)後に起こる。それゆえ**DistFreq**が0の場合は、disturbanceは回復後すぐに起こる。1の場合は回復時間の2倍の時間後にdisturbanceが起こり、スープが一杯になってしばらくの間空きメモリがほとんどない状態になる。

- **DistProp = .2** proportion of population affected by distrubance
  リーパーが起動したときこの値に後述の**SoupSize**を乗じたメモリーが空くまで、セルを殺す。殺されるセルはリーパーキューの先頭から、後述の**ReapRndProp**の確率で選択される。

- **DivSameGen = 0** cells must produce offspring of same genotype, to stop evolution
  この値が1のとき、もし娘セルのゲノムが自分と異なるとき細胞分裂(divide命令)を中断する。これは突然変異やsexにより進化が起こるをわざと防ぐときに用いる。

- **DivSameSiz = 0** cells must produce offspring of same size, to stop size change
  この値が1のとき、もし娘セルのゲノムサイズが自分と異なるとき細胞分裂を中断する。これはサイズの変化による進化をわざと防止する。

- **DropDead = 5** stop system if no reproduction in the last x million instructions
  突然変異率が高すぎる場合、すべてのセルが機能しなくなる場合がある。この値よりも長時間、新たなセルが生まれなかった場合システムを強制終了させる。

- **EjectRate = 0** rate at which random ejections from soup occur
  CM5または、ネットワークTierraでこの値が1以上にセットされていると、この値の個体数の分の1の確率で他のスープメモリーに娘セルが移住する。シングルプロセッサタイプのTierraでは同じ確率で死ぬことを意味する。

- **GenPerBkgMut = 32** mutation rate control by generations ("cosmic ray")
  バックグランド突然変異率("cosmic ray")の値。値32はそれぞれの世代でおよそ32個のセルのうち一つが1命令語の確率で突然変異する事を意味する。

- **GenPerFlaw = 32** flaw control by generations
  "flaw" 突然変異率です。値32はそれぞれの世代で、およそ32個体に1個体の確率で命令が1回不完全に実行される事を意味する。

- **GenPerMovMut = 8** mutation rate control by generations (copy mutation)
  命令語複製時の突然変異率。値8はそれぞれの世代でおよそ8個体に1個体の割合で、命令語が1個、5ビットのうちの1つのビットがランダムに変化してコピーされる。ただしこれは現在使用されていない。Tierra Version5.0では以下に述べるの突然変異率を代わりに用いる。

- **GenPerDivMut = 32**
  命令語単位の突然変異率です。数値の意味は**GenPerMovMut**と同じ。後述の**MutBitProp**により2種類の変異の仕方をする。

- **GenPerCroInsSamSiz = 32**
  Tierra Version5.0では交叉(crossingover)による変異が起こるようにした。同じゲノムサイズのセル同士で交叉が起こる率。

- **GenPerInsIns = 32**
  ゲノムの途中に別のコードが挿入される変異の率。

- **GenPerDelIns = 32**
  ゲノムの一部が失われる欠失という変異の率。

- **GenPerCroIns = 32**
  ゲノムのサイズが異なっても起こる交叉の率。

- **GenPerDelSeg = 32**
  ゲノムの一部（セグメント）が欠失する変異の率。セグメントとはnop命令で区切られたコード領域を言う。Tierraではジャンプやプロシッジャー・コールがnop命令で作られているテンプレートを目印にして行わる。従ってこのセグメント単位で変異をおこすことにより、機能単位のコード進化が起こりやすくなる事が予想される。

- **GenPerInsSeg = 32**
  セグメント単位での挿入の率。

- **GenPerCroSeg = 32**
  セグメント単位での交叉の率。

- **MutBitProp = .2** proportion of mutations that are bit flips
  命令語の複製エラーの仕方を設定する。命令語の複製エラーにはビットフリップによるものと、ある命令語がまったく別の命令語に入れ替わる2種類があります。前者は命令語5ビットのうちのあるビットが反転して別のある命令語になるのだが、その変わり方は命令セットのマッピングにより異なる。どのようにマッピングすると効率よく進化するかという問題はまだよく分かっていない。そのため、ビットフリップによる突然変異は効率が悪くなる可能性がある。後者は命令セットのマッピングに関係なく、ランダムに変異できるのでビットフリップより効率が良いと思われる。この値が0.2であるとは0.2の割合で前者のビットフリップ突然変異が起こり、0.8の割合で後者の突然変異が起こる事を意味する。

- **IMapFile = opcode.map** map of opcodes to instructions, file in GenebankPath
  GenebankPath の下にある命令セットのマップファイルの名前をセットする。

- **JmpSouTra = 0.** source track switches per average size
  JumpTrackProb = .2 probability of switching track during a jump of the IP
  これは**二倍体**以上のデジタル生物の実験を行うために設けられたパラメータである。二倍体以上では複数のCPUとコードを持つことができる。movii命令でコードをコピーするとき、それぞれのCPUはソーストラックと目的トラックが定義されている。このとき値**JmpSouTra**の確率でランダムにソーストラックと目的トラックが変化する。またCPUは実行トラックも定義できます。値**JumpTrackProb**の確率で命令ポインターが指すトラックを変化させる。

- **LazyTol = 10** tolerance for non-reproductive cells
  自己複製しないセルの個体数の許容値。

- **MalMode = 1** 0 = first fit, 1 = better fit, 2 = random preference,
  # 3 = near mother's address, 4 = near bx address
  # 5 = near top of stack address, 6 = suggested address (parse dependant)
  IBMのT.J.Watson Research CenterのChris Stephenson氏が作成したメモリーアロケーターのパラメーター。これは空きメモリーブロックを二分木で管理する。娘セルのメモリーを確保するときの方法を指定する。

  * **0 = first fit:**
    空きメモリーブロックを左（アドレスの若い方）から調べ、最初に充分なメモリーがあるブロックを確保する。

  * **1 = better fit:**
    これは二分木のrootから探索を始め、娘セル用メモリーと最も近い大きさのメモリーブロックを探索する。

  * **2 = random preference:**
    充分な大きさのメモリブロックをランダムに選択する。

  * **3 = near mother's address:**
    MapReapTol,MalTolで指定された許容値以内で、親セルのなるべく近くに娘セルのメモリブロックを確保する。
  * **4 = near bx address:**
    許容値以内の**bx**アドレス付近でメモリブロックを確保する。
  * **5 = near top of stack address:**
    許容値以内の先頭スタックの値で示されるアドレス付近でメモリブロックを確保する。
  * **6 = suggested address (parse dependent):**
    命令セットの復号の仕方で指定されたレジスタの値付近にメモリブロックを確保する。

- **MalReapTol = 1** 0 = reap by queue, 1 = reap oldest creature within MalTol
  娘セルの場所を考慮せずにリーパーが起動すると、セルの人口が増大するのが妨げられてしまう。そこで、この値を1にすると、**MalTol**の範囲でのみリーパーが古いセルを殺すようにできるのでそのような状態を防ぐことができる。

- **MalSamSiz = 0** force memory alloc to be same size as parent (stop evolution)
  この値を1にセットすると、親セルと同じサイズのメモリーアローケーションしか許さない。これは進化をわざと抑止する方法の一つである。

- **MalTol = 20** multiple of avgsize to search for free block
  「セルの平均ゲノムサイズ×この値」以内で空きメモリーを探索するようにする。

- **MateSizeEp = 2** size epsilon for potential mate
  交叉の相手を選ぶとき、ゲノムサイズの差がこの値以下のデジタル生物を探索する。

- **MaxCpuPerCell = 16** maximum number of CPUs allowed per cell
  並列性を実現するために、各デジタル生物は複数のCPUを有する事ができる。この値はその最大値。

- **MaxIOBufSiz = 8** maximum size for IOS buffer
  各セルのCPUにはIOバッファがある。その最大数をセットする。

- **MaxGetBufSiz = 4** maximum size for get IO buffer
  入力バッファの最大サイズ。

- **MaxPutBufSiz = 4** maximum size for put IO buffer
  出力バッファの最大サイズ。

- **MaxSigBufSiz = 8** maximum size for signal buffer
  CPUにはまたシグナルバッファというものも用意されている。この値はその最大サイズ。

- **MemModeFree = 0** read, write, execute protection for free memory
  MemModeMine = 0 rwx protection for memory owned by a creature
  MemModeProt = 2 rwx protection for memory owned by another creature
  # rwx protect mem: 1 bit = execute, 2 bit = write, 4 bit = read
  デジタル生物からみるとスープメモリの状態は3つの種類に区別される。すなわ

ち、自分が所有しているメモリ、他の生物が所有しているメモリ、だれにも所有されていないメモリ。それぞれの種類のメモリ状態に対して生物は読み取り、書き込み、実行の可否が区別される。このメモリ・プロテクションを1バイト(8ビット)で表現すると以下のようになる。

	execute	write	read
0	Y	Y	Y
1	N	Y	Y
2	Y	N	Y
3	N	N	Y
4	Y	Y	N
5	N	Y	N
6	Y	N	N
7	N	N	N

これらのメモリー・プロテクションの設定はコンパイル時に変更できる。

```
#define READPROT     /* define to implement read protection of soup */
#define WRITEPROT    /* define to implement write protection of soup */
#define EXECPROT     /* define to implement execute protection of soup */
```

デフォルトでは書き込み保護のみonになっている。他のプロテクション・モードに変更したい場合は、configur.hファイルを編集し、再コンパイルする。または、以下のパラメータでもプロテクション・モードの変更ができる。

* **MemModeFree**が0の場合、空きメモリー（どのデジタル生物にも所有されていないメモリー）はread,write,executeいずれも可能。
* **MemModeMine**が0の場合、デジタル生物は自分が所有しているメモリーに対して、read,write,executeいずれも可能。
* **MemModeProt**が2の場合、他のデジタル生物が所有しているメモリーに対して、read,executeはできますが、writeはできない。

- **MinCellSize = 12** minimum size for cells
  デジタル生物の最小のゲノムサイズを指定する。

- **MinGenMemSiz = 12** minimum size for genetic memory of cells
  細胞分裂(divide命令)実行時、娘セルのゲノムサイズがこの値より小さい場合、分裂できない。

- **MinTemplSize = 1** minimum size for templates
  相補的テンプレートの最小値。この値より小さいとテンプレートと認識されない。

- **MovPropThrDiv = .7** minimum proportion of daughter cell filled by mov
  divide命令実行時、娘セルのゲノムサイズが「この値×親セルのゲノムサイズ」より小さい場合、細胞分裂を中断する。このパラメータが必要な理由は、もしこれがないと、たくさんの空（ゲノムサイズ0）のセルだらけになってしまうからである。

- **new_soup = 1** 1 = this a new soup, 0 = restarting an old run
  この値が1の場合、Tierraは新規にシミュレーションを開始するものと見なす。Tierraを保存して終了すると、soup_outファイルにすべてのパラメータが保存される。そのとき、この値は0にセットされる。保存したデータから再スタートしたい場合は、Tierraの第1引数にsoup_outを指定する。

- **NumCells = 2** number of creatures and gaps used to inoculate new soup
  Tierra開始時に、スープメモリに置かれる先祖種の数を指定する。システムはこの値の数だけ、後述する先祖種のリストを読み込む。

- 現在スライサー機構は3種類(0-2番まで)の関数として用意されている。これらの関数はslicers.cというファイルにある。0番目のスライサーはSlicerQueue()という関数を使用する。これはゲノムサイズまたは定数値を用いてサイズ・スライスを計算する。1番目のスライサーはSlicerPhoton()という関数を使用する。これは光子と葉緑素の比喩を用いたスライサーである。2番目のスライサーはRanSlicerQueue()で固定部分にランダムな値をプラスして計算するスライサーである。後述する**SliceStyle**でどのスライサーを選択するかを設定する。

- **PhotonPow = 1.5** power for photon match slice size
  1番目のSlicerPhoton()をスライサーに選択した場合のパラメーター。後述の**SliceStyle**を1にセットした場合、CPU時間の割り当てを日光と光合成の比喩で行う。日光(光子)がスープメモリ上にランダムに降り注いでいることをイメージしてみる。ある命令語が葉緑素の役割を果たすとする。光子はある番号列として定義されている。光子がその命令語に当たると、光子の番号列と命令語の番号(機械語コード)がいくつマッチするかをカウントする。その値をその命令語を所有するセルのスライス・サイズとする。そしてそのスライス・サイズはこのパラメータの値で冪乗される。

- **PhotonWidth = 8** amount by which photons slide to find best fit
  光子がスープメモリにぶつかると、この値の1/2だけ左にスライドした位置からスタートし、この値の数だけ右に移動しながら葉緑素に相当する命令語の数を数える。

- **PhotonWord = chlorophill** word used to define photon
  葉緑素に相当する命令語のパターンを指定する。これは32を基とした付番システムになっている。数字の0から9までとアルファベットのaからvに番号を与える。アルファベットw,x,y,zは使用しない。そのためのchlorophyllのスペルが間違って綴られている。文字列の長さは79まで。

- **PutLimit = 10** distance for intercellular communication, mult of avg creat siz
  put/get命令ではIOバッファを使って、セル間でのコミュニケーションが可能である。コミュニケーションする相手はテンプレートを使って探す方法がある。そのとき、このパラメータでどれくらい離れた場所のテンプレートを探索するかを設定する。

- **ReapRndProp = .3** top prop of reaper que to reap from
  リーパーがセルを殺す方法を設定する。この値が0だと、リーパーはいつもリーパーキューの先頭にあるセルを殺す。もし1だと、リーパーはキューにあるセルをランダムに殺す。0.3の場合はキューの先頭30%にあるセルをランダムに殺す。

- **SearchLimit = 5** distance for template matching, mult of avg creat siz
  相補的テンプレートを探す最遠距離を指定。5というのは平均ゲノムサイズの5倍の距離を探索範囲とするという意味。実際の値は100万命令毎に更新される。

- **seed = 0** seed for random number generator, 0 uses time to set seed
  乱数のシード。0の場合はシステムを起動した時刻を使う。この時刻はtierra.logとsoup_outファイルに書き込まれるので、全く同じ条件でシミュレーションを繰り返したい場合に用いる事ができる。

- **SizDepSlice = 0** set slice size by size of creature
  2番目のスライサーRanSlicerQueue()を選択した場合のパラメーター。スライス・サイズ(*SizeSlice*)を固定部分とランダムな部分の和で計算する。この値が0だと以下の計算式を使ってスライス・サイズを計算する。

$$SizeSlice = SlicFixFrac \cdot SliceSize + rand((SlicRanFrac * SliceSize) + 1))$$

  SliceSize, SliceFixFrac, SliceRanFracは、後述する定数。またこの値が0でない場合はゲノムサイズを使って以下の式でスライス・サイズを計算する。

$$SizeSlice = SlicFixFrac \cdot SlicSiz + rand((SlicRanFrac * SlicSiz) + 1))$$

  ここで*SlicSiz*は$genome_siz^{SlicePow}$で計算される。

- **SlicePow = 1** set power for slice size, use when SizDepSlice = 1
  前述のように、**SizDepSlice**が1の場合に使用されるパラメータ。この場合ゲノムサイズをこの値で冪乗する。この値が1の場合、大きなサイズのゲノムにも小さなサイズのゲノムにも淘汰圧力は中立に働く。もし1よりも大きな値に設定されると、大きなサイズのゲノムに有利になる。もし1よりも小さな値だと、小さなサイズのゲノムに有利になる。

- **SliceSize = 25** slice size when SizDepSlice = 0
  SizDepSliceが0の場合に使用されるパラメータ。ゲノムサイズに関係なくこの値を基に、スライス・サイズを計算。

- **SliceStyle = 2** choose style of determining slice size
  前述したように、3つのスライサーのどれを使用するのかを設定する。

- **SlicFixFrac = 0** fixed fraction of slice size
  SliceStyleが2のとき、固定部分の計算式に用いる定数。

- **SlicRanFrac = 2** random fraction of slice size
  SliceStyleが2のとき、ランダム部分の計算式に用いる定数。

- **SoupSize = 60000** size of soup in instructions
  スープメモリの大きさ。いくつの命令語が置けるかをセットする。

- **MonPort = 17501**
  MigrCtrlPort = 17502
  Tierraの観測ツールBeagleと通信するときに使うポート番号。詳しくは**3.5節 Beagle**を参照。

- **center**
  0080aaa
  space 10000
  0045aaa
  先祖種の設定。この例では、スープメモリの真ん中**center**(SoupSize/2)に**0080aaa**を置き、さらに10000命令分の空白**space 10000**を空けて**0045aaa**を置くように指定している。

3.6 仮想CPU

Tierraの動作をよく理解するためには、仮想CPUの構造を理解しなければならない。ここではまず、Tierraのソースコードを基に、仮想CPUの仕組みを詳しく調べる。次に、仮想機械語命令がどのように実装されているかもソースコード（C言語）を見ながら解説する。

3.6.1 cell構造体

各セル（デジタル生物）はソースファイルのtierra.hに、以下のような構造体cellとして定義されている。

```
struct cell /* structure for cell of organisms */
{   Dem d;  /* fecundity and times and dates of birth and death */
    Que q;  /* pointers to previous and next cells in queues */
    Mem mm; /* main cell memory */
    Mem md; /* daughter cell memory */
    CpuA c; /* virtual cpus */
    I8s ld; /* 0 = dead, 1 = alive */
    ThrdAnaDat cell_thrdanadat; /* cell thread analysis data */
};
```

この構造体の中のCpuAというのが仮想CPUの入れ物である。次にこのCpuA構造体について説明する。

3.6.2 CpuA構造体

CpuAは以下のような構造体である。

```
typedef struct /* structure for cpu array */
{   TSignal sig;     /* signal buffer for this cell */
    I16s ib;         /* instruction bank */
    I16s ac;         /* number of this active cpu */
    Cpu *c;          /* pointer to currently active cpu */
    I16s n;          /* number of allocated cpus */
    Cpu *ar;         /* pointer to array of cpus */
    SyncA sy;        /* sync group array */
    InstDef *d;      /* pointer to current InstDef structure for parsing */
#ifdef NET
    IOb io;          /* IO buffer for network communications */
#endif /* NET */
```

```
#ifdef IO
    PGb pg;          /* IO buffer for put - get communications */
    GloCom gc;
#endif /* IO */
    I32s threadct; /* cell thread count */
} CpuA;
```

以下にこの構造体の主な要素について説明する。

- **I16s ib;** /* instruction bank */
  CPUがタイム・スライスをもらうと、命令バンク(instruction bank)の値がタイム・スライス分増加する。

```
ce->c.ib += size_slice;
```

  そして命令語を実行するたびに命令バンクの値は減少し、0になるとそのスライスが終了し、他のセルのCPUに実行が移る。

```
for(is.ts = ce->c.ib; is.ts > 0; )
{
    .....
    ce->c.ib -= is.dib;
    is.ts -= is.dib;
}
```

- **Cpu *ar;** /* pointer to array of cpus */
  I32s n; /* number of allocated cpus */
  マルチ・スレッド（多細胞）の機能を実現するために、CpuA構造体は仮想CPUを複数(n個)持つことができる。そしてそれは構造体CPUの配列へのポインターとして実装されている。構造体CPUについては後述する。

- **Cpu *c;** /* pointer to currently active cpu */
  I32s ac; /* number of this active cpu */
  現在動作中のCPUと、その番号をセットする。

- **InstDef *d;** /* pointer to current InstDef structure for parsing */
  命令語が復号（tierra.cの中の関数FetchDecode()が実行）されると、復号命令セットの構造体InstDefへのポインターがセットされる。構造体InstDefについては後述する。

```
I16s FetchDecode()
{   I16s di;
    .....
    is.eins = &soup[ce->c.c->ip];
    ......
    di = (*(is.eins)) % InstNum;
    ......
    ce->c.d = id + di;     <--- InstDef構造体へのポインターをセット
    ......
    (*id[di].decode) ();  <--- 復号関数へのポインタ
    ......
}
```

- **SyncA sy;** /* sync group array */
  セルが複数のCPUを持つ場合、構造体CpuAと構造体Cpu(後述)の要素syを使って、同期させることができます。これは命令語csync()で実行できる。

- **IOb io;** /* IO buffer for network communications */
  ネットワークバージョンでは、通信のためのIOバッファがある。このバッファはセルの中のすべてのCPUが共有する。このバッファはファイルinstruct.cの中の以下の関数により操作できる。

```
IOS * GetFreeIOS(cp) /* find an IOS to write incoming message to */
IOS * GetIOSOut(cp) /* get address of IOS output structure */
IOS * GetTagIOS(cp, tag) /* find next IOS with specified tag, to read */
void IncrIOSnio(cp) /* increment cp->c.io.nio with funky modulus */
I8s ReadIOS(cp, tag, dest) /* copy data from IOS buffer to soup */
I8s WriteIOS(cp, sou) /* copy data from soup to IOS buffer */
void InitIOS(ios) /* initialize IOS structure */
```

- **PGb pg;** /* IO buffer for put - get communications */
  GloCom gc;
  ネットワークバージョンでないTierraでは、スープメモリ内のセル同士の通信のために、以下のバッファが用意されている。こららのバッファも、ファイルinstruct.cの中の以下の関数により操作可能である。

```
Preg GetFreeGetBuf(cp) /* find get buf reg to write incoming message to */
Preg GetFreePutBuf(cp) /* find put buf reg to write incoming message to */
void put() /* write a value to the output buffer (prayer) */
void puticc() /* write a value to the output buffer */
I8s ReadFPut(cp, value) /* for god to read the data in the output buffer */
void get() /* read a value from the input buffer */
void Write2Get(cp, value) /* place value in input buffer of cell cp */
void Broad2Get(value) /* broadcast value to input buffer of all cells */
```

3.6.3 Cpu構造体

Cpu構造体もtierra.hに定義されている。

```
typedef struct {          /* structure for registers of virtual cpu */
    Reg re[ALOC_REG];     /* array of registers */
    Reg ip;               /* instruction pointer */
    Reg sp;               /* stack pointer */
    Reg st[STACK_SIZE];   /* stack */
    Flags fl;             /* flags */
    CRflags cf;           /* CPU Register flags */
    CSync sy;             /* CPU sync structure */
    I8s slicexit;         /* this cpu exits the current slice */
    I32s threadid;        /* thread id */
    I32s parthrdid;       /* parent thread id */
#if PLOIDY > 1
    I8s ex; /* track of execution */
    I8s so; /* source trace for reads */
    I8s de; /* destination track for writes */
    I8s wc; /* wait count for error-track switching */
#endif /* PLOIDY > 1 */
} Cpu;
```

以下にCpu構造体の各メンバーについて説明する。

- **Reg re[ALOC_REG];** /* array of registers */
  このCPUのレジスタ。ALOC_REGで定義された数のレジスタを持つ。ただし、"shadow referencing"という方式でレジスタを使用している場合は、実際に計算に使用できる数はその半分になる。"shadow referencing"とは、命令セット1で使用しているもので、数値演算命令に使用するレジスタを間接的に指定して、レジスタの順番を入れ換えられるようにする方式の事である。

- **Reg ip;** /* instruction pointer */
  命令ポインター。

- **Reg st[STACK_SIZE];** /* stack */
  サイズSTACK_SIZEのスタック。

- **Reg sp;** /* stack pointer */
  スタックポインター。

- **CSync sy;** /* CPU sync structure */
  前述のように、複数CPUを持つセルではcsync()関数を用いてセル内のすべてのCPUの同期をとることができる。2倍体以上のセルではどのコード・トラックを実行するのかを制御するために以下のようなメンバーが用意されている。

```
#if PLOIDY > 1
    I8s ex; /* track of execution */
    I8s so; /* source trace for reads */
    I8s de; /* destination track for writes */
    I8s wc; /* wait count for error-track switching */
#endif /* PLOIDY > 1 */
```

- **Flags fl;** /* flags */
  CPUには二つのフラグセットがある。flは、フラグ構造体Flagsを実装する。この構造体は命令語実行エラーなどに使用される。詳しくは次節で述べる。

- **CRflags cf;** /* CPU Register flags */
  Tierra Version4.1以降では各命令語で使われるレジスタの役割を、柔軟に切り替えるようになった。このフラグは各レジスタにどんな役割を果たさせるかを指定する。詳しくは次次項で述べる。

3.6.4 フラグ構造体Flags

フラグ構造体Flagsの中身は以下のようになっている。

```
typedef struct
{   int E:1;  /* Error : 1 = error condition has occurred
                          0 = no error condition occurred */
    int S:1;  /* Sign : resulting sign of arithmetic operation,
                        0 = +, 1 = - */
    int Z:1;  /* Zero : result of arithmetic or compare operation,
                        0 = nonzero, 1 = zero */
    int B:2;  /* Bits: 00 = 32, 01 = 16, 10 = 8 */
    int D:1;  /* Direction: for shift, rotate, move, search */
} Flags;
```

これらのフラグは命令語clrf()でクリアされる。

- **int E:1;** /* Error : 1 = error condition has occurred 0 = no error condition occurred */
  命令語が実行エラーのとき1にセットされるフラグ。

- **int S:1;** /* Sign : resulting sign of arithmetic operation,0 = +, 1 = - */
  数値演算の結果、その値の符号によりセットされるフラグ。

- **int Z:1;** /* Zero : result of arithmetic or compare operation, 0 = nonzero, 1 = zero */
  数値演算の結果、値が0になったか、比較演算の結果が真だったとき1にセットされる。

- **int B:2;** /* Bits: 00 = 32, 01 = 16, 10 = 8 */
  命令語movdi(),movid()で移動されるビットの数を制御するために使用される。以

下の命令語で、移動ビット数を 8,16 または 32 の間で切り替えます。命令語 togbf() で切り替え可能。

- **int D:1;** /* Direction: for shift, rotate, move, search */
  shift,rotate,move そして search 関係の命令の方向を制御するのに使用される。命令語 togdf() で切り替え可能。

3.6.5 CRflags 構造体

構造体 **CRflags** は以下のように定義されている。

```
typedef struct
{   TRind Se;   /* current segment register */
    TRind De;   /* current destination register */
    TRind So;   /* current  source register */
} CRflags;      /* CPU register flags */

typedef struct
{   I8s i; /* index into Cpu.re array of registers */
    I8s t; /* index into Rtog.r list of registers */
} TRind;                          /* register toggle indexes */
    /* note that TRind.t will be toggled by incrementing it with the
       modulus Rtog.n, after each toggle, TRind.i will be updated to
       provide a more direct reference to the actual Cpu.re register */
```

Tierra 実行開始時、後述する **GetAMap()** 関数が opcode.map ファイルを読み込むとき、そのファイルの各行のはじめの文字列が Se,De,So のいずれかの文字になっていないかを調べる。もし、そうならそれらの行の内容を復号し、それぞれ segment（ゲノムサイズ、カウンタ用）,destination（コピー先アドレス用）, source（コピー元アドレス用）レジスタのリストとして以下のグローバルな構造体データ IDRegs にセットする。

```
IDflags idf;  /* flags usage */

typedef struct /* there will be only one global copy of this structure */
{   Rtog Se; /* list of segment registers */
    Rtog De; /* list of destination registers */
    Rtog So; /* list of source registers */
} IDRegs;

typedef struct
{   I8s n;  /* number of register to toggle */
    I8s *r; /* list of n registers to toggle */
} Rtog;     /* for keeping track of toggled registers */
```

どのレジスタも以下の命令語を用いて、segment,destination, source レジスタに切り替える事ができる。

```
void toger() /* toggle segment registers */
void togdr() /* toggle destination registers */
void togsr() /* toggle source registers */
```

これらの関数は、CRflags 構造体の値を切り替える。用途が切り替えられたレジスタは、例えば decode.c の中の movdi() 命令で次のコードのように使用される。

```
    {   if (ce->c.d->idf.De)  /* using destination toggle registers */
            tval0 = flaw() + ce->c.c->cf.De.i;
        else    /* using fixed destination registers provided by opcode.map */
            tval0 = flaw() + ce->c.d->re[0];
        if (ce->c.d->idf.So)  /* using source toggle registers */
            tval1 = flaw() + ce->c.c->cf.So.i;
        else    /* using fixed source registers provided by opcode.map */
            tval1 = flaw() + ce->c.d->re[1];
        if (ce->c.d->idf.Se)  /* using segment toggle registers */
            tval2 = flaw() + ce->c.c->cf.Se.i;
        else
            tval2 = flaw() + ce->c.d->re[2];
    }
```

3.7 命令セット

CPUは,「命令の読み込み (Fetch)」→「復号 (Decode)」→「実行 (execute)」→「IPのインクリメント (IncrementIp)」という一連のサイクルを繰り返す。IPによって指し示されている機械語命令がCPUに読み込まれ、そのビット・パターンを解読（復号）して対応する命令を特定し、その命令を実行する。そしてIPの値は次に実行すべき命令のアドレス値になるようインクリメントされる。これはスライサー関数から呼ばれるタイムスライス関数TimeSlice()(tierra.c)によって行われている。

```
void TimeSlice(size_slice)
    I32s size_slice;
{   I16s c, di;
        .........
        di = FetchDecode(); /* Fetch and decode instruction */
        .........
        (*id[di].execute)(); /* Execute instruction */
        .........
        IncrementIp();  /* Increment ip */
        .........
        SystemWork();
        .........
}
```

3.7.1 実行時命令セットの設定

Tierraはバージョン4.1以降では、実行時に命令セットを指定できるようになった。以前は、命令セットを変更する毎にソースコードを再コンパイルしなければならなかったそうだ。

実行時に命令セットを設定できるようにするため、ジーンバンク・ディレクトリの下にあるopcode.mapというファイルを使用する。このファイル名はスープファイル(si0,si1...))の中でIMapFileというパラメーターで設定されている。Tierra起動時、ソースコードファイルgenio.cの中の関数GetAMap()が、このopcode.mapファイルの内容を読み込む。そしてその内容が以下のような構造体配列InstDef *idにセットされる。

```
typedef struct /* structure for instruction set definitions */
{   I8s op;                     /* op code */
    I32s cyc;                   /* the cost in cpu cycles of this instruct */
    I8s mn[9];                  /* assembler mnemonic */
    void (*execute) P_((void)); /* pointer to execute function */
    void (*decode) P_((void));  /* pointer to decode function */
    I8s re[8];                  /* register assignments */
    IDflags idf;                /* flags usage */
} InstDef;

InstDef *id;
```

次項でこのInstDef構造体について説明する。

3.7.2 InstDef構造体

- **I8s op;** /* op code */
  命令語のオペコード、すなわちスープメモリからゲノムコードとして得た値で、機械語命令に復号するための値。現在のTierraではこれは0から255までの値をとる。

- **I32s cyc;** /* the cost in cpu cycles of this instruct */
  Tierra Version5.0で新しく導入されたフィールドで、命令語を実行するときに必要なCPUサイクル数を設定する。

- **I8s mn[9];** /* assembler mnemonic */
  命令語のアセンブラ・ニーモニックです。アセンブラ"arg"はASCII形式のファイルを読み込み、そこに書かれているニーモニックをアセンブルしてバイナリファイルを作成する。

- **void (*execute) P_((void));** /* pointer to execute function */
  この命令が実際に行う関数へのポインター。各命令語の関数はinstruct.cに実装されている。GetAMap()関数がopcode.mapファイルを読み込むとき、opcode.mapファイル内のこのフィールドは無視される。

- **void (*decode) P_((void));** /* pointer to decode function */
  この命令を復号する関数へのポインター。復号関数はdecode.cに実装されている。GetAMap()関数がopcode.mapファイルを読み込むとき、opcode.mapファイル内のこのフィールドは無視される。

- **I8s re[8];** /* register assignments */
  ほとんどの命令語は演算の対象となる値をレジスタから読み取り、結果をそのままたは別のレジスタに格納する。配列re[8]はsourceレジスタ（演算対象）とdestinationレジスタ（演算結果用）を指定する。opcode.mapファイルには、アルファベット文字列でこれを記述する。例えば、文字'a'は0番のレジスタ、'b'は1番のレジスタを指定する。命令セット0のopcode.mapファイルでは、例えばsubAAC命令は以下のようになっている。

```
{0, 1, "subAAC", math, dec1d2s, "aac", ""}, /* "aac" */
```

  文字列"aac"とあるが、最初の'a'がdestination register、次の二文字'ac'はsource registerになる。

- **IDflags idf;** /* flags usage */
  フラグセットのビットフィールド。opcode.mapの各行の最後の要素である。

```
{0, 1, "movdd", movdd, dec1d1s, "ab", "H"},   /* "rr" */
```

  この例では、'H'という文字でフラグセットが指定されている。これは"shadow register"を使用するためのフラグを設定する。構造体IDflagsについて次節で説明する。

3.7.3 IDflags構造体

IDflags構造体は、以下のように定義されている。

```
typedef struct
{   unsigned Se:1;  /* Offset Segment F -> O */
    unsigned B:1;   /* Bits */
    unsigned De:1;  /* Destination T -> D */
    unsigned So:1;  /* Source      G -> S */
    unsigned D:1;   /* Direction   D -> I */
    unsigned H:1;   /* sHadow registers */
    unsigned P:1;   /* reverse Polish */
    unsigned C:1;   /* speCial */
} IDflags;
```

- **unsigned Se:1;** /* Offset Segment F - > O */

  segmentレジスタの切り替えを可能にするフラグ。decode.cの中でif(ce- > c.d- > idf.Se)というコードで使用されている。

- **unsigned B:1;** /* Bits */
  このフラグがセットされると、レジスタとスープメモリの間でデータを移動するとき、移動量の単位を1byte,2byte,4byteのうち、どれかを選択するか設定する。ソースではce- > c.c- > fl.Bというコードで選択している。movdi命令では、以下のように使用されている。

```
void movdi() /* move from soup to register, e.g.: R0 = soup [R1] */
{   I8s  i8, *pi8s, *pi8d;
    I16s i16;
    I32s NumBytes, i;
    Ints i32;

    ce->c.c->fl.E = ce->c.c->fl.S = ce->c.c->fl.Z = 0;
    switch (ce->c.d->mn[5])
    {   case 0:    /* movdi() */
        {   if (ce->c.d->idf.B)
            {   switch (ce->c.c->fl.B)
                {   case 0:
                        goto byte4;
                    case 1:
                        goto byte2;
                    /* case 2: default to byte1; */
                }
            }
            NumBytes = 1;
            pi8d = &i8;
            break;
        }
        case '2':  /* movdi2() */
        {   byte2:
            NumBytes = 2;
            pi8d = (I8s *) &i16;
            break;
        }
        case '4':  /* movdi4() */
        {   byte4:
            NumBytes = 4;
            pi8d = (I8s *) &i32;
            break;
        }
    }
```

  転送単位を何byteにするかは、togbf関数で切り替えられる。

- **unsigned De:1;** /* Destination T - > D */

  destinationレジスタの切り替えを可能にするフラグ。decode.cファイルの中で、if(ce- > c.d- > idf.De)というコードで用いられている。

- **unsigned So:1;** /* Source G - > S */

  sourceレジスタの切り替えを可能にするフラグ。decode.cファイルの中で、if(ce- > c.d- > idf.So)というコードで用いられている。

- **unsigned D:1;** /* Direction D - > I */
  shift,rotate,move,search 命令の方向を制御するフラグ。現在は使用していない。

- **unsigned H:1;** /* sHadow registers */
  命令セット1ではレジスタを間接的に参照する方法が取り入れられている。“shadow register” を用いることにより、どの命令語もどのレジスタによっても使用できるようになる。フラグビットフィールドに、‘H’ とセットするだけで、この shadow register を使用することができる。

- **unsigned P:1;** /* reverse Polish */
  命令セット2では、逆ポーランド記法を使用している。これはある命令でレジスタの順番をシフトアップまたはシフトダウンさせる必要がある。フラグビットフィールドに‘P’ とセットするだけで、この機能を使用できる。

- **unsigned C:1;** /* speCial */
  re[8] フィールドで指定した、レジスタ文字列だけでは定義できない、複雑な source,destination 関係を必要とする命令語がある。そのような命令語の場合、このフラグをセットする。

3.7.4 GetAMap() 関数

GetAMap() 関数が opcode.map ファイルを読み込むとき、そのファイルの中に2個より多くの命令語がセットされていれば、この関数はまず opcode.map は2ビットの命令セットであると認識する。そして命令語の数が2の倍数になる度に、命令セットの大きさを1ビットずつ（つまり2倍に）拡張する。もし、命令語の数は2の倍数でない場合（例えば、命令語が25しか記述されてない場合、次の2の倍数32に対して7語足りない。）、この関数は足りない分をファイルの先頭行から読み直して順番に埋めていく。

opcode.map ファイルは構造体 InsDef を反映したものである。GetAMap() 関数は、このファイルの各行の3つのダブルコーテーションで囲まれた文字列を読み込む。

```
{0, 1, "movdd", movdd, dec1d1s, "ab", "H"},   /* "rr" */
```

“movdd” は mn[9] に、“ab” は re[8] に、“H” は IDflags に代入される。オペコード op はその読み取られた順に 0x00 から付番される。

soup_in.h ファイルは idt という名前の InstDef 構造体を定義している。これは、Tierra で使用できるすべての命令語で構成されたスーパーセットとなっている。idt で定義された構造体要素のうち、以下の3つが実行時命令セットを確定するのに使用される。

```
I8s mn[9];                  /* assembler mnemonic */
void (*execute) P_((void)); /* pointer to execute function */
void (*decode) P_((void));  /* pointer to decode function */
```

GetAMap() 関数が opcode.map ファイルを読み取る度に、このファイルのアセンブラニーモニックと、idt で定義されているニーモニックを突き合わせ、命令語の実行関数と、復号関数をセットする。

3.7.5 利用可能な命令語、idt[] 配列

InstDef構造体配列idtは、利用可能な命令語のすべてを定義する。この配列の中から実行時に使用する命令セットをopcode.mapファイルで選択する。現在のidt配列はsoup_in.hファイル内に以下のように定義されている。

```
InstDef idt[] =
{   {0, 1, "nop0", nop, pnop, "", {0}},  /* no decode args */
    {0, 1, "nop1", nop, pnop, "", {0}},  /* no decode args */
    {0, 1, "add", add, dec1d2s, "", {0}},  /* "rrr" */
    {0, 1, "add2", add, dec1d2s, "", {0}},  /* "rrr" */
    {0, 1, "adrb", adr, decadr, "", {0}},  /* "rr  " */
    {0, 1, "adrf", adr, decadr, "", {0}},  /* "rr  " */
    {0, 1, "adro", adr, decadr, "", {0}},  /* "rr  " */
    {0, 1, "and", and, dec1d2s, "", {0}},  /* "rrr" */
    {0, 1, "and2", and, dec1d2s, "", {0}},  /* "rrr" */
    {0, 1, "call", tcall, ptcall, "", {0}},  /* no decode args */
    {0, 1, "clrf", clrf, pnop, "", {0}},  /* no decode args */
    {0, 1, "clrfi", clrfi, pnop, "", {0}},  /* no decode args */
    {0, 1, "clrrf", clrrf, pnop, "", {0}},  /* no decode args */
    {0, 1, "csync", csync, pnop, "", {0}},  /* no decode args */
    {0, 1, "dec", add, dec1d1s, "", {0}},  /* "rr" */
    {0, 1, "dec2", add, dec1d1s, "", {0}},  /* "rr" */
    {0, 1, "dec4", add, dec1d1s, "", {0}},  /* "rr" */
    {0, 1, "decC", add, dec1d1s, "cc", {0}},  /* "cc" */
    {0, 1, "div", idiv, dec1d2s, "", {0}},  /* "r" */
    {0, 1, "div2", idiv, dec1d2s, "", {0}},  /* "r" */
    {0, 1, "divide", divide, dec2s, "", {0}},  /* "rr" */
    {0, 1, "enter", enter, pnop, "", {0}},  /* no decode args */
    {0, 1, "exch", exch, dec2s, "", {0}},  /* "rr" */
    {0, 1, "getregs", getregs, dec1s, "t", {0}},  /* "#" */
    {0, 1, "halt", halt, pnop, "", {0}},  /* no decode args */
    {0, 1, "ifE", skip, dec2s, "", {0}},  /* "rr" */
    {0, 1, "ifS", skip, dec2s, "", {0}},  /* "rr" */
    {0, 1, "ifZ", skip, dec2s, "", {0}},  /* "rr" */
    {0, 1, "ifequal", skip, dec2s, "", {0}},  /* "rr" */
    {0, 1, "ifgrtr", skip, dec2s, "", {0}},  /* "rr" */
    {0, 1, "ifless", skip, dec2s, "", {0}},  /* "rr" */
    {0, 1, "ifsig", skip, dec2s, "", {0}},  /* "rr" */
    {0, 1, "ifz", skip, dec2s, "", {0}},  /* "rr" */
    {0, 1, "inc", add, dec1d1s, "", {0}},  /* "rr" */
    {0, 1, "incA", add, dec1d1s, "aa", {0}},  /* "aa" */
    {0, 1, "incB", add, dec1d1s, "bb", {0}},  /* "bb" */
    {0, 1, "incC", add, dec1d1s, "cc", {0}},  /* "cc" */
    {0, 1, "jmpb", adr, decjmp, "", {0}},  /* "r" */
    {0, 1, "jmpf", adr, decjmp, "", {0}},  /* "r" */
    {0, 1, "jmpo", adr, decjmp, "", {0}},  /* "r" */
    {0, 1, "join", join, pnop, "", {0}},  /* no decode args */
    {0, 1, "mal", malchm, dec1d3s, "", {0}},  /* "rr r" */
    {0, 1, "movBA", movdd, dec1d1s, "ba", {0}},  /* "ba" */
    {0, 1, "movDC", movdd, dec1d1s, "dc", {0}},  /* "dc" */
    {0, 1, "movdd", movdd, dec1d1s, "", {0}},  /* "rr" */
    {0, 1, "movdi", movdi, pmovdi, "", {0}},  /* "rr" */
    {0, 1, "movdi2", movdi, pmovdi, "", {0}},  /* "rr" */
    {0, 1, "movdi4", movdi, pmovdi, "", {0}},  /* "rr" */
    {0, 1, "movid", movid, pmovid, "", {0}},  /* "rr" */
    {0, 1, "movid2", movid, pmovid, "", {0}},  /* "rr" */
    {0, 1, "movid4", movid, pmovid, "", {0}},  /* "rr" */
    {0, 1, "movii", movii, pmovii, "", {0}},  /* "rr" */
    {0, 1, "movii2", movii, pmovii, "", {0}},  /* "rr" */
    {0, 1, "movii4", movii, pmovii, "", {0}},  /* "rr" */
    {0, 1, "mul", mul, dec1d2s, "", {0}},  /* "r" */
    {0, 1, "mul2", mul, dec1d2s, "", {0}},  /* "r" */
    {0, 1, "not", not, dec1d, "", {0}},  /* "r" */
    {0, 1, "notl", notl, dec1d, "", {0}},  /* "rr" */
    {0, 1, "not0", not0, dec1d, "", {0}},  /* "rr" */
    {0, 1, "offAACD", offset, dec1d3s, "aacd", {0}},  /* "aacd" */
    {0, 1, "offBBCD", offset, dec1d3s, "bbcd", {0}},  /* "bbcd" */
    {0, 1, "offset", offset, dec1d3s, "", {0}},  /* "rrrr" */
    {0, 1, "ior", ior, dec1d2s, "", {0}},  /* "rrr" */
    {0, 1, "ior2", ior, dec1d2s, "", {0}},  /* "rrr" */
    {0, 1, "pop", pop, dec1d, "", {0}},  /* "r" */
    {0, 1, "popA", pop, dec1d, "a", {0}},  /* "a" */
    {0, 1, "popB", pop, dec1d, "b", {0}},  /* "b" */
    {0, 1, "popC", pop, dec1d, "c", {0}},  /* "c" */
```

```
    {0, 1, "popD", pop, dec1d, "d", {0}},  /* "d" */
    {0, 1, "popE", pop, dec1d, "e", {0}},  /* "e" */
    {0, 1, "popF", pop, dec1d, "f", {0}},  /* "f" */
    {0, 1, "push", push, dec1s, "", {0}},  /* "r" */
    {0, 1, "pushA", push, dec1s, "a", {0}},  /* "a" */
    {0, 1, "pushB", push, dec1s, "b", {0}},  /* "b" */
    {0, 1, "pushC", push, dec1s, "c", {0}},  /* "c" */
    {0, 1, "pushD", push, dec1s, "d", {0}},  /* "d" */
    {0, 1, "pushE", push, dec1s, "e", {0}},  /* "e" */
    {0, 1, "pushF", push, dec1s, "f", {0}},  /* "f" */
    {0, 1, "rand", movdd, dec1d1s, "", {0}},  /* "rr" */
    {0, 1, "ret", pop, dec1d, "", {0}},  /* no decode args */
    {0, 1, "rolld", rolld, pnop, "", {0}},  /* no decode args */
    {0, 1, "rollu", rollu, pnop, "", {0}},  /* no decode args */
    {0, 1, "shl", shl, dec1d, "", {0}},  /* "r" */
    {0, 1, "shr", shr, dec1d, "", {0}},  /* "r" */
    {0, 1, "slicexit", slicexit, pnop, "", {0}},  /* no decode args */
    {0, 1, "split", split, dec1d1s, "", {0}},  /* "rr" */
    {0, 1, "stup", stup, pnop, "", {0}},  /* no decode args */
    {0, 1, "stdn", stdn, pnop, "", {0}},  /* no decode args */
    {0, 1, "sub", add, dec1d2s, "", {0}},  /* "rrr" */
    {0, 1, "sub2", add, dec1d2s, "", {0}},  /* "rrr" */
    {0, 1, "subAAC", add, dec1d2s, "aac", {0}},  /* "bac" */
    {0, 1, "subBAC", add, dec1d2s, "bac", {0}},  /* "bac" */
    {0, 1, "subCAB", add, dec1d2s, "cab", {0}},  /* "cab" */
    {0, 1, "subCBA", add, dec1d2s, "cba", {0}},  /* "cba" */
    {0, 1, "subCCD", add, dec1d2s, "ccd", {0}},  /* "ccd" */
    {0, 1, "surf", migrate, dec1s, "", {0}},  /* "r" */
    {0, 1, "surff", migrate, dec1s, "", {0}},  /* "r" */
    {0, 1, "togbf", togbf, pnop, "", {0}},  /* no decode args */
    {0, 1, "togdf", togdf, pnop, "", {0}},  /* no decode args */
    {0, 1, "toger", toger, pnop, "", {0}},  /* no decode args */
    {0, 1, "togsr", togsr, pnop, "", {0}},  /* no decode args */
    {0, 1, "togdr", togdr, pnop, "", {0}},  /* no decode args */
    {0, 1, "ttime", ttime, dec1d, "", {0}},  /* no decode args */
    {0, 1, "xor", xor, dec1d2s, "", {0}},  /* "rrr" */
    {0, 1, "xor2", xor, dec1d2s, "", {0}},  /* "rrr" */
    {0, 1, "zero", movdd, dec1d1s, "", {0}},  /* "rr" */
    {0, 1, "zeroD", movdd, dec1d1s, "dd", {0}},  /* "dd" */
#ifdef IO
    {0, 1, "get", get, dec1d1s, "", {0}},  /* "r" */
    {0, 1, "put", put, dec1d1s, "", {0}},  /* "rr" */
    {0, 1, "puticc", puticc, pputicc, "", {0}},  /* "rr" */
#endif                                  /* IO */
#if PLOIDY > 1
    {0, 1, "trso", trso, pnop, "", {0}},  /* no decode args */
    {0, 1, "trde", trde, pnop, "", {0}},  /* no decode args */
    {0, 1, "trex", trex, pnop, "", {0}},  /* no decode args */
#else                                   /* PLOIDY > 1 */
    {0, 1, "trso", NULL, NULL, "", {0}},
    {0, 1, "trde", NULL, NULL, "", {0}},
    {0, 1, "trex", NULL, NULL, "", {0}},
#endif                                  /* PLOIDY > 1 */
#ifdef SHADOW
    {0, 1, "A", regorder, dec1s, "a", {0}},  /* "a" */
    {0, 1, "B", regorder, dec1s, "b", {0}},  /* "b" */
    {0, 1, "C", regorder, dec1s, "c", {0}},  /* "c" */
    {0, 1, "D", regorder, dec1s, "d", {0}},  /* "d" */
#else                                   /* SHADOW */
    {0, 1, "A", NULL, NULL, "", {0}},
    {0, 1, "B", NULL, NULL, "", {0}},
    {0, 1, "C", NULL, NULL, "", {0}},
    {0, 1, "D", NULL, NULL, "", {0}},
#endif                                  /* SHADOW */
#ifdef NET
    {0, 1, "tpings", tpingsnd, dec1s, "", {0}},  /* "r" */
    {0, 1, "tpingr", tpingrec, dec1d, "", {0}},  /* "r" */
    {0, 1, "getip", getip, dec1d, "", {0}},  /* "r" */
    {0, 1, "getipp", getipp, dec1d, "", {0}},  /* "r" */
    {0, 1, "getippf", getipp, dec1d, "", {0}},  /* "r" */
#endif                                  /* NET */
    {0, 1, "END", NULL, NULL, "", {0}},  /* this line must be last! */
};
```

これから上記の各命令語のうちのいくつかについて説明するが、命令語で共通に使用されている操作がいくつかあるので、それらについて先に説明する。

3.7.6 DoRPNd(),DoRPNu() 関数

命令セット2では逆ポーランド記法を使用している。算術演算が行われた後、DoRPNd() 関数を使ってレジスタの内容をボトムからトップに向かって順番にポップする。また、データがレジスタに格納されると、DoRPNu() 関数を使ってトップのレジスタから下位のレジスタに順番にデータがプッシュする。これらの操作は、'P' フラグがセットされた場合に有効になる。

```
void DoRPNd()
{   I16s i;

    if (ce->c.d->idf.P)
        for (i = 1; i < NUMREG - 1; i++)
            ce->c.c->re[i] = ce->c.c->re[i + 1] + flaw();
}

void DoRPNu()
{   I16s i;

    if (ce->c.d->idf.P)
        for (i = NUMREG - 1; i > 0; i--)
            ce->c.c->re[i] = ce->c.c->re[i - 1] + flaw();
}
```

3.7.7 DoFlags() 関数

それぞれの命令語が実行されたあと、'E','S','Z' フラグが0にクリアされる。ただし、destination レジスタの値が0の場合はZフラグは1に、負の場合は、Sフラグが1にセットされる。命令が失敗したときは、Eフラグが1にセットされる。しかしifE,ifS,ifZ命令ではこの操作は行われない。

```
void DoFlags()
{   ce->c.c->fl.E = ce->c.c->fl.S = ce->c.c->fl.Z = 0;
    if (*(is.dreg) == 0)
        ce->c.c->fl.Z = 1;
    if (*(is.dreg) < 0)
        ce->c.c->fl.S = 1;
}
```

3.7.8 DoMods(),DoMods2(),DoMods3() 関数

もし復号関数がグローバル変数 PInst is の要素 is.dmod が0ではない値にセットした場合、destination レジスタの値は剰余関数 mo() を使用して、その値を0から is.dmod-1 の範囲に収めるようにする。また復号関数が is.dran の値を0以外の値にセットした場合、destination レジスタの値は、-is.dran から is.dran の範囲になければならない。もしオーバーしていた場合は0になる。これらの操作の後、is.dmod, is.dran は0にリセットされる。これは、多くの算術命令で使用される。演算結果がスープメモリーサイズの上限値を越えないようにするためなどに利用している。

```
void DoMods()
{   if (is.dmod)
        *(is.dreg) = mo(*(is.dreg), is.dmod);
    else if (is.dran && (*(is.dreg) > is.dran || *(is.dreg) < -is.dran))
        *(is.dreg) = 0;
    is.dmod = is.dran = 0;
}
```

destination レジスタが複数ある場合、以下の関数が用いられる。

```
void DoMods2()
{   if (is.dmod2)
        *(is.dreg2) = mo(*(is.dreg2), is.dmod2);
    else if (is.dran2 && (*(is.dreg2) > is.dran2 || *(is.dreg2) < -is.dran2))
        *(is.dreg2) = 0;
    is.dmod2 = is.dran2 = 0;
}

void DoMods3()
{   if (is.dmod3)
        *(is.dreg3) = mo(*(is.dreg3), is.dmod3);
    else if (is.dran3 && (*(is.dreg3) > is.dran3 || *(is.dreg3) < -is.dran3))
        *(is.dreg3) = 0;
    is.dmod3 = is.dran3 = 0;
}
```

以上述べた3つ関数は、復号関数がグローバル変数Pinst isに値をセットする事で正しく機能する。そのデータ構造は以下のようになっている。

```
typedef struct  /* struct for passing arguments from decode to execute */
{   Preg sreg;  /* pointer to source register */
    FpIns sins; /* pointer to source instruction */
    I32s sval;  /* source value */
    I8s stra;   /* source track */

    Preg dreg;  /* pointer to destination register */
    I32s dval;  /* original destination value */
    FpIns dins; /* pointer to destination instruction */
    I8s dtra;   /* destination track */
    I32s dmod;  /* destination modulused positive this size */
    I32s dran;  /* destination kept in signed range of this value */
    Pcells dcel; /* destination cell */

    Preg sreg2; /* pointer to 2nd source register */
    FpIns sins2;/* pointer to 2nd source instruction */
    I32s sval2; /* 2nd source value */
    I8s stra2;  /* 2nd source track */

    Preg dreg2; /* pointer to 2nd destination register */
    I32s dval2; /* original destination value */
    FpIns dins2;/* pointer to 2nd destination instruction */
    I8s dtra2;  /* 2nd destination track */
    I32s dmod2; /* 2nd dest modulused positive this size */
    I32s dran2; /* 2nd dest kept in signed range of this value */

    I32s sval3; /* 3rd source value */
    I32s dval3; /* original destination value */

    Preg dreg3; /* pointer to 3rd destination register */
    I32s dmod3; /* 3rd dest modulused positive this size */
    I32s dran3; /* 3rd dest kept in signed range of this value */

    I8s mode;   /* mode of instruction */
    I8s mode2;  /* 2nd mode of instruction */
    I8s mode3;  /* 3rd mode of instruction */

    I8s expr;   /* execute protection 0 = no protection, 1 = protected */
    I32s oip;   /* address of instruction being executed: ce->c.ip */
    I32s oipofst; /* ip offset in cell */
    I16s oncpu; /* number of allocated cpus */
    I16s oac;   /* number of this active cpu */
    I32s othreadid; /* thread id */
    Dem odem; /* cell demographic data */
    Mem ocellmem;
#if PLOIDY > 1
    I32s oextrk; /* execution track */
#endif /* PLOIDY > 1 */
    I8u odstoffsettype; /* 0-in cell, 1-daughter, 2-free mem */
    I32s odstoffset; /* destination offset */
#ifdef NET
    N32u onodeip;
    I8u  onodetype;
#endif /* NET */
    FpIns eins; /* pointer to instruction being executed */
    I16s  di;   /* decoded instruction (current)  for debugging */
    I16s pdi;   /* decoded instruction (previous) for debugging */
    I8s iip;    /* amount to increment instruction pointer */
    I8s dib;    /* amount to decrement instruction bank */
```

```
    I16s ts;    /* size of time slice, used to control central loop */
} PInst;
```

一般的に、復号関数は演算の元になるデータをグローバル変数 is の要素 sval,sval2,sval3 に置き、使用するレジスタへのポインターを dreg,dreg2 などにセットする。つまり is は復号関数と実行関数の間でデータを受け渡すのに使用される。

3.7.9 命令語

以下に idt[] 配列の中の主要な命令語について説明していく。

- **{0, 1, "nop0", nop, pnop, "", {0}},** /* no decode args */
  {0, 1, "nop1", nop, pnop, "", {0}}, /* no decode args */
  これら二つは命令ポインターのインクリメント以外 no-ops（何もしない）命令語。しかしながら、adrb,adrf,adro,call,jmpb,jmpf,jmpo などのパターンマッチングを必要とする命令語のテンプレートとして使用される。こちらの命令が失敗する条件はない。
- **{0, 1, "add", add, dec1d2s, "", {0}},** /* "rrr" */

```
 * is.dreg  = destination register, where calculation will be stored
 * is.sval  = a value that will be added to is.sval2 and placed in dest reg
 * is.sval2 = a value that will be added to is.sval  and placed in dest reg
 * is.dmod  = value by which to modulus destination register
 * is.dran  = range within which to contain destination register
```

  add 命令は二つのソースデータを加算し、結果を destination レジスタに置く。source と destination レジスタは InstDef の 6 番目フィールドで指定される。"flaw"(命令語にわざと欠陥を与える) が復号関数 dec1d2s() で復号されるときに 1 番目のソースデータに与えられる。DoMods(),DoFlags(),DoRPNd() で計算結果のチェックが行われる。

- **{0, 1, "add2", add, dec1d2s, "", {0}},** /* "rrr" */
  add2 命令は復号ステージでの操作だけが異なる。この命令は "De","Se"' フラグがセットされ、destination レジスタもまたソースデータを持っていたりするようなトリッキーな場合に用いられる。本質的にこの命令は 3 つではなく、2 つレジスタを用いて加算演算する場合に使用され、そのうちの 1 つまたは両方が "De" と "Do" フラグでレジスタの役割が特定される。
- **{0, 1, "adrb", adr, decadr, "", {0}},** /* "rr " */

```
* is.mode  = search mode: 1 = forward, 2 = backward, 0 = outward
 * is.mode2 =  preference: 1 = forward, 2 = backward, and return for
 *         direction found: 1 = forward, 2 = backward, 3 = both, 0 = none
 * is.dval  = starting address for forward search
 * is.dval2 = starting address for backward search
 * is.dreg  = destination register where target address will be stored
 * is.dreg2 = destination register where template size will be stored
 * is.dreg3 = destination register where offset of target will be stored
 * is.sval  = return address if template size = 0
 * is.sval2 = template size, 0 = no template
 * is.sval3 = search limit, and return for distance actually searched
 * is.dmod  = modulus value for is.dreg
 * is.dmod2 = modulus value for is.dreg2
```

```
* is.dmod3 = modulus value for is.dreg3
* is.dran  = range to maintain for is.dreg
* is.dran2 = range to maintain for is.dreg2
* is.dran3 = range to maintain for is.dreg3
```

この命令語は、相補的テンプレートを後方探索（スープメモリのアドレスが減る方向に探索）する。探索できたアドレスを、最初のdestinationレジスタにセットする。テンプレートはこの命令語のすぐ後のnop0,nop1命令列である。このテンプレートをつくるnop命令の個数をテンプレートサイズとして、2番目のdestinationレジスタにセットする。マッチングされる相補的テンプレート（ターゲットテンプレート）の方は、元のテンプレート（ソーステンプレート）より大きくてもかまわない。ソーステンプレートとマッチする部分列が含まれているテンプレートが見つかると、マッチしたnop命令の部分列のアドレス+1が1番目のdestinationレジスタにセットされる。

この命令は以下の4つのレジスタを使う。レジスタの指定はInstDef構造体の6番目のフィールドでセットできる。

1. ターゲット・テンプレートのアドレスをセットするためのdestinationレジスタ
2. テンプレートサイズをセットするためのdestinationレジスタ
3. ソース・テンプレートのアドレスからターゲット・テンプレートアドレスまでのオフセットをセットするためのdestinationレジスタ
4. ターゲット・テンプレートを探索するための始点とソース・テンプレートとのオフセットが格納されたsourceレジスタ。

この命令語は4つのレジスタを使用するが、レジスタを全く使用しない方法も可能。'C'フラグをセットする事により、InstDefで指定されていないレジスタを使用不能にする。例えば、

```
{0, 1, "divide", divide,  dec2s, "ac", ''C''},  /* "rr  " */
```

では、reフィールドは'a','c'レジスタのみを指定しているので、'a'レジスタはターゲットテンプレートのアドレスを、'c'レジスタはテンプレート・サイズがセットされる。

テンプレートの最初のアドレスをAとし、テンプレートサイズをSとすると、後方探索開始アドレスはA-S-1になる。探索はターゲットテンプレートが見つかるか、探索限界(**SearchLimit**)に達するまで行われる。探索アドレスがスープアドレスの最後に達すると、スープアドレスの0番から探索を再開する。

また命令ポインターがソース・テンプレートの分(S+1)移動する。

この命令はソース・テンプレートサイズが**MinTemplSize**より小さいか、**SoupSize**より大きいか、またはターゲット・テンプレートが探索範囲**SearchLimit**内に見つからなかった場合、失敗となる。失敗の場合、**cx**レジスタは変化しない。DoMods(),DoMods2(),DoMods3() で値のチェックが行われる。

- **{0, 1, "adrf", adr, decadr, "", {0}}, /* "rr " */**
  {0, 1, "adro", adr, decadr, "", {0}}, /* "rr " */
  adrbと同様な命令だが、adrfは前方探索、adroは両方向探索を行う。前方探索はアドレスA+S+1から始まる。

- **{0, 1, "and", and, dec1d2s, "", {0}}, /* "rrr" */**

```
* is.dreg  = destination register, where calculation will be stored
* is.sval  = value that will be anded by is.sval2 and placed in dest reg
* is.sval2 = value that will be anded into is.sval and placed in dest reg
* is.dmod  = value by which to modulus destination register
* is.dran  = range within which to contain destination register
```

and命令は二つのソース・バリューをビットごとにANDし、その結果をdestinationレジスタに格納する。復号関数dec1d2s()により第1ソース・バリューに"flaw"が作用する。

この命令は第2ソース・バリューが0だと失敗する。DoMods(),DoFlags(),DoRPNs()による値チェックがある。

- **{0, 1, "and2", and, dec1d2s, "", {0}}, /* "rrr" */**
  and2命令は復号ステージでの操作だけが異なる。この命令は"De","Se"'フラグがセットされ、destinationレジスタもまたソースデータを持っていたりするようなトリッキーな場合に用いらる。本質的にこの命令は3つではなく、2つレジスタを用いて加算演算する場合に使用され、そのうちの1つまたは両方が"De"と"Do"フラグでレジスタの役割が特定される。

- **0, 1, "call", tcall, ptcall, "", 0,** /* no decode args */

```
   * void adr() find address of a template
 * is.mode  = search mode: 1 = forward, 2 = backward, 0 = outward
 * is.mode2 =  preference: 1 = forward, 2 = backward, and return for
 *        direction found: 1 = forward, 2 = backward, 3 = both, 0 = none
 * is.dval  = starting address for forward search
 * is.dval2 = starting address for backward search
 * is.dreg  = destination register where target address will be stored
 * is.dreg2 = destination register where template size will be stored
 * is.dreg3 = destination register where offset of target will be stored
 * is.sval  = return address if template size = 0
 * is.sval2 = template size, 0 = no template
 * is.sval3 = search limit, and return for distance actually searched
 * is.dmod  = modulus value for is.dreg
 * is.dmod2 = modulus value for is.dreg2
 * is.dmod3 = modulus value for is.dreg3
 * is.dran  = range to maintain for is.dreg
 * is.dran2 = range to maintain for is.dreg2
 * is.dran3 = range to maintain for is.dreg3
 *
 * void push()
 * is.sval = value to be pushed onto the stack
```

この命令はテンプレートマッチングを使ってプロシッジャー呼びだしを行う。命令ポインターIPがスタックにpushされ、そしてそれはターゲットテンプレートの次のアドレスにジャンプする。この命令はadr()とpush()命令を組み合わせて行われる。

- **{0, 1, "clrf", clrf, pnop, "", {0}},** /* no decode args */ この命令は仮想CPUのフラグセットCpu.flをクリアする。

```
void clrf()                         /* clear all Cpu.Flags */
{   ce->c.c->fl.E = ce->c.c->fl.S = ce->c.c->fl.Z =
        ce->c.c->fl.B = ce->c.c->fl.D = 0;
}
```

- **{0, 1, "clrfi", clrfi, pnop, "", {0}},** /* no decode args */ この命令はフラグセットCpu.Flagsの CPUサイクルに関するフラグをクリアする。

```
void clrfi()              /* clear Cpu.Flags associated with CPU cycles */
{   ce->c.c->fl.E = ce->c.c->fl.S = ce->c.c->fl.Z = 0;
}
```

- **{0, 1, "clrrf", clrrf, pnop, "", {0}},** /* no decode args */ この命令はCpu.CRflagsをクリアする。

```
void clrrf()                        /* clear all Cpu.CRflags */
{   ce->c.c->cf.Se.t = 0;
    ce->c.c->cf.Se.i = IDregs.Se.r[0];
    ce->c.c->cf.De.t = 0;
    ce->c.c->cf.De.i = IDregs.De.r[0];
    ce->c.c->cf.So.t = 0;
    ce->c.c->cf.So.i = IDregs.So.r[0];
    ce->c.c->fl.E = ce->c.c->fl.S = ce->c.c->fl.Z = 0;
}
```

- **{0, 1, "csync", csync, pnop, "", {0}},** /* no decode args */
この命令は一つのセルに含まれているすべてのCPUの同期を取る（多細胞生物の場合）。あるひとつのCPUがこの命令を実行すると、他のすべてのCPUがこの命令を実行するまでのそのCPUは停止する。そして、すべてのCPUがこの命令の次の命令から、一斉に再スタートする。

- **{0, 1, "getregs", getregs, dec1s, "t", {0}},** /* "#" */
この命令は、同じセル内の他のCPUのレジスタの値を"get"する。この命令はセルが一つしかCPUを持っていない場合は失敗する。InstDefのreフィールドに0,1または2を指定して、どのCPUからデータをもらうかの方式を指定する。0と指定すると、CPU配列での一個前のCPUからデータをもらう。1と指令すると、CPU配列での一個後のCPUからデータをもらう。2と指定すると、このCPU以外の別のCPUをランダムに選択してデータをもらう。

- **{0, 1, "halt", halt, pnop, "", {0}},** /* no decode args */ CPUを停止し、セルから削除する。もし、最後の1個のCPUがこの命令を実行すると、セル自身が活動を止める。

- **{0, 1, "join", join, pnop, "", {0}},** /* no decode args */
多細胞生物用の命令。この命令を実行すると、そのセルの最初のCPU以外は停止

し削除される。もしセルの最初のCPUがこの命令を実行すると、他のすべてのセルがこの命令を実行するまで待つ。

- **{0, 1, "split", split, dec1d1s, "", {0}}, /* "rr" */**

```
* is.sval = source value for processor number
* is.dval = destination register # for new processor number
```

多細胞生物用の命令。この命令はセルの中に新しいCPUを生成する。Cpu構造体の中身（レジスタ、命令ポインター、スタック、スタックポインタ、フラグその他）が新しいセルにコピーされる。ただし一つのレジスタの値だけ変化します。そのレジスタはInstDef構造体のreフィールドで指定される。オリジナルおCPUのあるレジスタの値は2倍され、新しいCPUのレジスタの値は2倍されたあとに1加算されます。この仕組みは一つのセルの中でCPUを識別するID番号を与えるための機構です。

この命令は、セル内のCPUの数がスープパラメータの中の**MaxCpuPerCell**を越えるとエラーになる。

- **{0, 1, "surf", migrate, dec1s, "", {0}}, /* "r" */**
  {0, 1, "surff", migrate, dec1s, "", {0}}, /* "r" */
  {0, 1, "tpings", tpingsnd, dec1s, "", {0}}, /* "r" */
  {0, 1, "tpingr", tpingrec, dec1d, "", {0}}, /* "r" */
  {0, 1, "getip", getip, dec1d, "", {0}}, /* "r" */
  {0, 1, "getipp", getipp, dec1d, "", {0}}, /* "r" */
  {0, 1, "getippf", getipp, dec1d, "", {0}}, /* "r" */
  ネットワークTierraのための命令語。詳しい説明は省略。

- **{0, 1, "END", NULL, NULL, "", {0}}, /* this line must be last! */**
  InstDef idtを定義するとき、このラインが最後に来なければならない。

このidt[]配列の中からいくつかの命令語を選んで命令セットを作り、ジーンバンクファイルのopcode.mapに記述する。次節以降で代表的な命令セット0から3までについて解説する。

3.7.10 命令セット0の特徴

この命令セットではスープメモリのアドレスを保持するためにAX,BXの2つのレジスタを使用する。その値は0 ～ SoupSizeの間の数値を取るように管理されます。CX,DXは正負どちらの値も保持できるがその最小値と最大値は-SoupSizeから+SoupSizeまでに制限される。もしこの値をオーバーするとその値は強制的に0にされてしまう。

実行結果はレジスタには反映されない。通常、命令が実行されるたびに命令バンクのデクリメントされる。また分岐命令(jmp,adrf,callなど)以外は命令ポインターをインクリメントする。

命令が成功した場合、フラグレジスタはクリア(値0に)されます。失敗すると、destinationレジスタへの値の格納は行われず、フラグレジスタにエラーがセットされる。

以下に命令セット0の32個の命令語のうち前節で解説しなかったものについて説明する。

- **{0, 1, "not0", not0, dec1d, "", {0}}**, /* "rr" */
  CXレジスタの最下位ビットを反転する。

- **{0, 1, "shl", shl, dec1d, "", {0}}**, /* "r" */
  CXレジスタの全ビットを1つずつ左シフトする。最下位ビットは0になる。この命令も失敗する条件はない。

- **{0, 1, "zero", movdd, dec1d1s, "", {0}}**, /* "rr" */
  CXレジスタの値を0にする。

- **{0, 1, "ifz", skip, dec2s, "", {0}}**, /* "rr" */
  CXレジスタの値が0の場合、命令ポインターの値を1インクリメントし、CXレジスタの値が0でな場合、命令ポインターの値を2インクリメントする。

- **{0, 1, "subCAB", add, dec1d2s, "cab", {0}}**, /* "cab" */
  AXレジスタの値からBXレジスタの値を減算し、CXレジスタにセットする。

- **{0, 1, "subAAC", add, dec1d2s, "aac", {0}}**, /* "bac" */
  AXレジスタの値からCXレジスタの値を減算し、AXレジスタにセットする。

- **{0, 1, "incA", add, dec1d1s, "aa", {0}}**, /* "aa" */
  {0, 1, "incB", add, dec1d1s, "bb", {0}}, /* "bb" */
  {0, 1, "incC", add, dec1d1s, "cc", {0}}, /* "cc" */
  それぞれ、AX,BX,CXレジスタの値に1を加える。

- **{0, 1, "decC", add, dec1d1s, "cc", {0}}**, /* "cc" */
  DXレジスタの値から1を減じる。

- **{0, 1, "pushA", push, dec1s, "a", {0}}**, /* "a" */
  {0, 1, "pushB", push, dec1s, "b", {0}}, /* "b" */
  {0, 1, "pushC", push, dec1s, "c", {0}}, /* "c" */
  {0, 1, "pushD", push, dec1s, "d", {0}}, /* "d" */
  それぞれ、AX,BX,CX,DXレジスタの値をスタックにプッシュする。そしてスタックポインターの値を1インクリメントする。(STACK_SIZEの剰余が取られる。)

- **{0, 1, "popA", pop, dec1d, "a", {0}}**, /* "a" */
  {0, 1, "popB", pop, dec1d, "b", {0}}, /* "b" */
  {0, 1, "popC", pop, dec1d, "c", {0}}, /* "c" */
  {0, 1, "popD", pop, dec1d, "d", {0}}, /* "d" */
  それぞれスタックのトップからAX,BX,CX,DXレジスタに値をポップする。そして

スタックポインタの値を1デクリメントする。(STACK_SIZEの剰余が取られます。)

- **{0, 1, "jmpo", adr, decjmp, "", {0}}**, /* "r" */
  この命令につづくソーステンプレートと相補的にマッチするターゲットテンプレートを探して、命令ポインターをその直後にセットする。

  このジャンプ命令は、両方向探索を行う。ソーステンプレートの開始アドレスをAとし、そのソーステンプレートのサイズをSとすると、前方探索はアドレスA+S+1から始まり、後方探索はA-S-1から始まる。探索は1ステップずつ、前方、後方にターゲットテンプレートが見つかるか、探索限界Searchlimitに到達するまで交互に行われる。

  この命令は、ソーステンプレートのサイズがMinTemplSizeより小さいか、SoupSizeより大きいか、ターゲットテンプレートが探索できないと失敗する。失敗の場合は命令ポインターはソーステンプレートの最後尾の直後(ip+S+1)を指すようにセットされる。

- **{0, 1, "jmpb", adr, decjmp, "", {0}}**, /* "r" */
  jmpoと同様だが、この命令は後方探索のみを行う。

- **{0, 1, "ret", pop, dec1d, "", {0}}**, /* no decode args */
  スタックのトップの値を命令ポインターにポップし、スタックポインターの値を1デクリメントする(STACK_SIZEで剰余演算される)。これにより、call命令で呼び出された直後の位置にリターンする。

- **{0, 1, "movDC", movdd, dec1d1s, "dc", {0}}**, /* "dc" */
  CXレジスタの値をDXレジスタにコピーする。

- **{0, 1, "movBA", movdd, dec1d1s, "ba", {0}}**, /* "ba" */
  AXレジスタの値をBXレジスタにコピーします。

- **{0, 1, "movii", movii, pmovii, "", {0}}**, /* "rr" */
  AXレジスタの値で指されるスープアドレスにある命令語をBXレジスタの値で指されるスープアドレスにコピーする。

  この命令は、(a)AXとBXの値が同じ場合、(b)BXレジスタで指されるスープアドレスがセルのものでない場合、(c)AXレジスタで指されるスープアドレスがセルのものでなくて、かつ読みと保護されている場合、そしてどちかの(d)アドレスがスープメモリからオーバーしていた場合、に失敗する。

- **{0, 1, "mal", malchm, dec1d3s, "", {0}}**, /* "rr r" */
  この命令はスープメモリの割り当てを要求する。要求メモリスペースの値はCXレジスタにセットされていなければならない。割り当てられたメモリの先頭アドレスがAXレジスタにセットされる。そしてそのメモリに対してsoup_inパラメー

タのMemModeMine, MemModeProtで指定される保護機構がセットされる。この命令に成功するとセルはリーパーキューの順位が一つ下がる。

Tierraオペレーティングシステムは要求されたサイズの空きメモリーブロックを探索する。もし、見つからなかった場合、リーパーが起動する。メモリーの探索方法はsoup_inパラメーターの所で説明したMalMode,MalReapTol,MalTolによって変わる。リーパーは要求されたメモリブロックが見つかるまで。リーパーによって他のセルを殺す。しかし、この命令を実行したセル自身は殺さない。

この命令は、(a)要求メモリのサイズが0の場合。(b)セルが2つ目のメモリブロックを要求した場合。つまりmal命令後、divide命令を行わずにもう一度mal命令を行おうとした場合。(c)要求したメモリサイズが「MaxMalMult×セルサイズ」より大きい場合、に失敗となる。

- **{0, 1, "divide", divide, dec2s, "", {0}}, /* "rr" */**
  セルが娘セルのメモリブロックへの書き込み特権を失い、娘セルが独立する。娘セルはリーパーキューの最後尾に入る。またスライサーキューのこのセルの次の位置に入ります。この命令が成功すると、セルのリーパーキューも順位が一つ下がる。

  この命令が失敗する条件は以下の通り。(a) 娘セルのサイズがsoup_inパラメーターのMinCellSizeより小さい。(b) 実際に書き込まれた娘セルのコードサイズce->d.mov_daughtがsoup_inパラメーターの¡MovPropThrDivより小さい。(c)soup_inパラメーターのDivSameSizeが0でなくて、娘セルのサイズがセルのサイズと異なる。(d)soup_inパラメーターのDivSameGenが0でなくて、娘セルのサイズとコード内容ががセルのと異なる。

3.7.11 命令セット1～3の特徴

命令セット1から3までは、レジスタ間のデータの受渡し方法以外は本質的にみな同じである。以下にその特徴を述べる。

- 命令セット1は自由にレジスタの順番を並べ変えることができる。そしてmovdd()命令で2つのレジスタの間で数値を直接移動させることができる。命令セット2は逆ポーレンド記法を用いてレジスタをロールアップしたり、ロールダウンできる。そして下位の2つのレジスタの間でデータを交換できる。命令セット3はスタックを通してレジスタ間のデータを移動する。

- また、movdi()命令でレジスタからスープにデータを移動したり、movid()命令でスープからレジスタにデータを移動できる。

- put(),get()命令でI/Oバッファにデータを読み書きできる。get()命令はインプットバッファからデータを読み、レジスタに移動させる。入力バッファはバッファから次に読み込まれるデータへのポインタ、次に書き込まれるデータのポインタ、そしてまだ読み込まれていないデータの個数の管理を行う。

```
ce->c.gb[GETBUFSIZ]     == pointer to next input value to be read
ce->c.gb[GETBUFSIZ + 1] == pointer to next input value to be written
ce->c.gb[GETBUFSIZ + 2] == number of unread input values
```

出力バッファは次に書き込まれるデータへのポインタ、次に読み込まれるデータのポインタそしてまだ読み込まれていないデータの個数の管理を行う。

```
ce->c.pb[PUTBUFSIZ]     == pointer to next output value to be written
ce->c.pb[PUTBUFSIZ + 1] == pointer to next output value to be read
ce->c.pb[PUTBUFSIZ + 2] == number of unread output values
```

put() 命令は実装の仕方により、2つのバージョン（put(),puticc()）がある。puticc() は、他のセルの入力バッファにデータを書き込む。puticc() 命令の後にテンプレートがあると、この命令は探索範囲 (PutLimit) 内で、get() 命令とそれにつづく相補的テンプレートを持ったセルを探し、そのセルの入力バッファに書き込む。もし puticc() 命令の後にテンプレートがないと、適当なレジスタの値をアドレスとして、そのアドレスのスープメモリ位置にあるセルの入力バッファに書き込む。

ただの put() 命令は自分自身の出力バッファに書き込みを行う。これは、Tierra を使用するユーザーさんたちがデジタル生物たちとコミュニケーションできるようにと用意された。ユーザーにとって有益な仕事をさせるためににデータを与え、セルはそのデータも使って計算し、その結果を出力させる事が可能である。このようなデジタル生物とユーザーとのコミュニケーションのために、以下の3つの関数が instruct.c に用意されている。

```
I8s ReadFPut(ce, value)   /* for god to read the data in the output buffer */
void Write2Get(ce, value) /* place value in input buffer of cell ce */
void Broad2Get(value)     /* broadcast value to input buffer of all cells */
```

- 命令セット1～3のすべての命令セットでは基本的な演算命令：inc(), dec(), add(), sub() が用意されている。zero() 命令はレジスタの値を0にする。not0() 命令はレジスタの最下位ビットを反転するが、not() 命令はレジスタの値のすべてのビットを反転する。命令セット2ではさらに rand() 命令で AX レジスタに乱数をセットする。

- レジスタの値が0かどうかを調べる ifz() 命令の他に ifZ() 命令がある。これはフラグがセットされているかどうかを調べて、直前に行った命令が成功か失敗かを判断する。

- 命令セット1から3では、jmpo(),jmpb() 命令は相補的テンプレートへのジャンプを行う他に、もしソーステンプレートがない場合は特定の指定されたレジスタの値をアドレスとしてそこにジャンプする。

- 命令セット1から3では、adro(),adrb(),adrf() 命令は2つの新しい特徴を持つ。1) ソーステンプレートとターゲットテンプレート間のオフセットを返す。2) ソーステンプレートにレジスタで指定されたオフセット値を加えた所から探索を開始します。これは、最初に探したテンプレートより、次はもっと遠くにあるテンプレートを探させたい場合などに使用できる。

- 命令セット1から3では、mal() 命令はレジスタの値を使ってsoup_inパラメーターのパラメータMalModeで述べた“better fit”モードで、空きメモリを探索できる。

- 命令セット1から3では、divide() 命令は母セルのレジスタ内容を娘セルにコピーする他に、命令ポインタの値を母セルが指定するあるレジスタをオフセットとして、自分のアドレスに加えてセットすることができる。これは母セルが娘セルのコードを一部分しか実行させないようにすることにより「細胞分化」を起こす事を狙ったものである。

3.8 先祖種ファイル

スープファイルの最終行に先祖種を指定するが、それは以下のようなASCII形式のアセンブラコード・ファイル（ゲノムファイル）として用意されなければならない。以下は命令セット0(ジーンバングgb0)の先祖種のゲノムファイル0080aaa.tieである。

```
***** blank line at the top of file *****

format: 3  bits: 2156009669  EXsh    TCsh    TPs     MFs     MTd     MBh
genotype: 0080aaa  genetic: 0,80  parent genotype: 0666god
1st_daughter:  flags: 0  inst: 827  mov_daught: 80          breed_true: 1
2nd_daughter:  flags: 0  inst: 809  mov_daught: 80          breed_true: 1
Origin: InstExe: 0,0  clock: 0  Thu Jan 01 -5:00:00 1970
MaxPropPop: 0.8306  MaxPropInst: 0.4239 mpp_time: 0,0
ploidy: 1  track: 0
; comments: the ancestor, written by a human, mother of all other creatures.

CODE

track 0:

nop1    ; 110 01   0 beginning marker
nop1    ; 110 01   1 beginning marker
nop1    ; 110 01   2 beginning marker
nop1    ; 110 01   3 beginning marker
zero    ; 110 04   4 put zero in cx
not0    ; 110 02   5 put 1 in first bit of cx
shl     ; 110 03   6 shift left cx (cx = 2)
shl     ; 110 03   7 shift left cx (cx = 4)
movDC   ; 110 18   8 move cx to dx (dx = 4)
adrb    ; 110 1c   9 get (backward) address of beginning marker -> ax
nop0    ; 100 00  10 complement to beginning marker
nop0    ; 100 00  11 complement to beginning marker
nop0    ; 100 00  12 complement to beginning marker
nop0    ; 100 00  13 complement to beginning marker
subAAC  ; 110 07  14 subtract cx from ax, result in ax
movBA   ; 110 19  15 move ax to bx, bx now contains start address of mother
adrf    ; 110 1d  16 get (forward) address of end marker -> ax
nop0    ; 100 00  17 complement to end marker
nop0    ; 100 00  18 complement to end marker
nop0    ; 100 00  19 complement to end marker
nop1    ; 100 01  20 complement to end marker
incA    ; 110 08  21 increment ax, to include dummy instruction at end
subCAB  ; 110 06  22 subtract bx from ax to get size, result in cx
nop1    ; 110 01  23 reproduction loop marker
nop1    ; 110 01  24 reproduction loop marker
nop0    ; 110 00  25 reproduction loop marker
nop1    ; 110 01  26 reproduction loop marker
mal     ; 110 1e  27 allocate space (cx) for daughter, address to ax
call    ; 110 16  28 call template below (copy procedure)
nop0    ; 100 00  29 copy procedure complement
nop0    ; 100 00  30 copy procedure complement
nop1    ; 100 01  31 copy procedure complement
nop1    ; 100 01  32 copy procedure complement
divide  ; 110 1f  33 create independent daughter cell
jmpo    ; 110 14  34 jump to template below (reproduction loop)
nop0    ; 100 00  35 reproduction loop complement
nop0    ; 100 00  36 reproduction loop complement
```

```
nop1    ; 100 01  37 reproduction loop complement
nop0    ; 100 00  38 reproduction loop complement
ifz     ; 000 05  39 dummy instruction to separate templates
nop1    ; 110 01  40 copy procedure template
nop1    ; 110 01  41 copy procedure template
nop0    ; 110 00  42 copy procedure template
nop0    ; 110 00  43 copy procedure template
pushA   ; 110 0c  44 push ax onto stack
pushB   ; 110 0d  45 push bx onto stack
pushC   ; 110 0e  46 push cx onto stack
nop1    ; 110 01  47 copy loop template
nop0    ; 110 00  48 copy loop template
nop1    ; 110 01  49 copy loop template
nop0    ; 110 00  50 copy loop template
movii   ; 110 1a  51 move contents of [bx] to [ax] (copy one instruction)
decC    ; 110 0a  52 decrement cx (size)
ifz     ; 110 05  53 if cx == 0 perform next instruction, otherwise skip it
jmpo    ; 110 14  54 jump to template below (copy procedure exit)
nop0    ; 110 00  55 copy procedure exit complement
nop1    ; 110 01  56 copy procedure exit complement
nop0    ; 110 00  57 copy procedure exit complement
nop0    ; 110 00  58 copy procedure exit complement
incA    ; 110 08  59 increment ax (address in daughter to copy to)
incB    ; 110 09  60 increment bx (address in mother to copy from)
jmpo    ; 110 14  61 bidirectional jump to template below (copy loop)
nop0    ; 100 00  62 copy loop complement
nop1    ; 100 01  63 copy loop complement
nop0    ; 100 00  64 copy loop complement
nop1    ; 100 01  65 copy loop complement
ifz     ; 000 05  66 this is a dummy instruction to separate templates
nop1    ; 110 01  67 copy procedure exit template
nop0    ; 110 00  68 copy procedure exit template
nop1    ; 110 01  69 copy procedure exit template
nop1    ; 110 01  70 copy procedure exit template
popC    ; 110 12  71 pop cx off stack (size)
popB    ; 110 11  72 pop bx off stack (start address of mother)
popA    ; 110 10  73 pop ax off stack (start address of daughter)
ret     ; 110 17  74 return from copy procedure
nop1    ; 100 01  75 end template
nop1    ; 100 01  76 end template
nop1    ; 100 01  77 end template
nop0    ; 100 00  78 end template
ifz     ; 000 05  79 dummy instruction to separate creatur
```

3.8.1 ヘッダー

ゲノムファイルはいくつかのヘッダー情報から始まる。以下にこのヘッダー情報について説明する。

- ******* blank line at the top of file *******
  ゲノムファイルの第1行目は空行にする。

- **format: 3**
  ゲノムファイルのフォーマットは過去に何度か変更された。以前はアセンブラはこのフォーマット番号によって、以降のゲノムファイルの解析を行っていた。現在は、これとそっくり同じように指定する。

- **bits: 2156009669**
  ジーンバンクのそれぞれのゲノムに関係付けられたビットフィールド。スープファイルのパラメーター GeneBnker と、WatcthExe,WatchMov,WatchTem のどれかが0でない値にセットされていた場合、このゲノタイプの生態的特徴によってビットフィールドがセットされる。セットの仕方は tierra.h の g_list 構造体のメンバー (I32u bits) と **3.5節 soup_in パラメーター** の WatchExe の項で定義されている。

- **EXsh TCsh TPs MFs MTd MBh**
  ビットフィールドをASCII形式で表示したもの。

- **genotype: 0080aaa**
  ゲノタイプの名前。名前の最初の数字4桁がゲノムサイズ。後のアルファベット3桁が、他の同じサイズのゲノムと区別するためにユニークに付与される文字列である。

- **genetic: 0,80**
  ゲノムのスープメモリ上のデフォルトの初期位置と、ゲノムサイズ。

- **parent genotype: 0666god**
  このゲノムの親のゲノタイプ名。0080aaaは神であり、また悪魔でもあるトム・レイ自身によって作成された（＾＾；）。ジーンバンク機能によってこのファイルが作成される場合、自動的に親ゲノム名が記述される仕掛けになっている。ただしこの機能は今の所、不完全である。親ゲノムがジーンバンクに永久登録されるためのしきい値に達っしていなかったりすると、この部分は不明になってしまう。そして、その親のゲノム名は再利用されてしまうため、たとえこの部分にゲノム名が記述されていたとしても、それが本物の親かどうかわからない。

- **1st_daughter: flags: 0 inst: 827 mov_daught: 80 breed_true: 1**
  このゲノムによって最初の娘セルが作られる過程での“代謝データ”を表示する。flagsは命令がエラーだった回数。instは一つ目の娘セルを作るのに必要だった命令実行回数。mov_daughtは娘セルにコピーした命令語の数。breed_trueは、これが1なら娘セルは自分と同じゲノタイプとして複製された事を意味する。

- **2nd_daughter: flags: 0 inst: 809 mov_daught: 80 breed_true: 1**
  このゲノムによって2番目のセルが作られる過程での“代謝データ”を表示する。普通2回目以降の自己複製では、“自己検査”は行わない事が多く、1回目と2回目で命令実行回数などが異なる。

- **Origin: InstExe: 0,0 clock: 0 Thu Jan 01 -5:00:00 1970**
  このゲノタイプが生まれた時刻を表示。InstExeはTierraシミュレータの時間（全命令実行回数。コンマの左側が100万命令実行時間ごとにカウントアップされる。右側は100万命令以下の回数。）。clockはシステムクロック時間。ここではC言語の時間関数の起点が記述されている。

- **MaxPropPop: 0.8306 MaxPropInst: 0.4239 mpp_time: 0,0**
  このゲノタイプが最も人口が多かったときの割合と、そのときのスープメモリの占有割合、およびその時の時刻（Tierra全命令実行回数）。

- **ploidy: 1 track: 0**
  ploidyは倍数体のレベル数（ゲノムの本数）。これは1なので半数体（ゲノムが1本だけ)である。trackは生まれたとき、どのトラック（ゲノム）のコードから実行するのかを示す。これは2倍体以上でなければ意味がない。

3.8.2 コード本体

以下にコード本体の説明をする。

- **CODE**
  ヘッダーが終わって、コードが開始する。

- **track 0:**
  ゲノムのトラック番号。この下からがトラック0のコードであるという宣言である。もし2倍体ならトラック0のコードの下に、track 1:と宣言してトラック1のコードを書く。

- **nop1 ; 110 01 0 beginning marker**
  実際のゲノムの最初の行。nop1は命令語のアセンブラ・ニーモニック。

  セミコロンから右側はコメントである。“110”はこの命令を実行したデジタル生物が何であるかの記録になっている。3桁の10進数で、1桁目（一番左の桁）は自分自身でこの命令を実行したことがある事を示す。2桁目は他のセル（パラサイト）がこの命令を実行した事を示す。3桁目は現在使用されていない。このコメントはスープファイル・パラメータWatchExeがセットされていた場合に表示される。

  つぎの“01”は、この命令語のオペコードの16進数表示である。スープメモリにはこの値が格納される。

  最後のコメント“0”は、この命令語のこのコードにおける順番である。0番から始まる。

3.8.3 Writing a Creature

もし、自分自身で先祖種のコードを作成したい場合は、以下の形式に従うこと。

```
**** begin genome file (note blank line at top of file)

format: 3  bits: 3
genotype: 0080aaa  parent genotype: 0666god

CODE

track 0:

nop1    ;
nop1    ;
**** end genome file
```

ゲノムファイルは以下の形式を守って作成しなければならない。

1. 最初の行は空行にする。

2. 上記のようにformatとbitsを指定する。

3. ゲノム名を指定する。ゲノム名のサイズの部分は、コードの命令語数と一致していなければならない。3桁のアルファベットでは26文字のうちどんな組合せでもかまわない。しかし、“aaa”を使用するのが好ましい。なぜならジーンバンカーは

このアルファベットを元にゲノム名の記憶領域を確保するので、aaaが最もメモリ使用効率が良い。

4. parent genotypeは実際には使用されていないが、適当に正しい形式のゲノム名を与えて置くこと。

5. 1行空けて、"CODE"と記した行を入れる。

6. また1行空けて、trackを記述した行を入れる。

7. また1行空けて、実際のコードを書き始める。コードはopcode.mapファイルで定義されている命令セットからアセンブラ・ニーモニックで記述する。

8. ファイル名は"ゲノム名.tie"とする。

3.9 ALmond Tool

Tierra Version4.3までのソースコードにはALmond Tool (Artificial Life Monitor)のソースコードが含まれていた。これはTierraとは独立したプログラムで、口絵1のようなスープメモリのデジタル生物の様子をビジュアルに表示するものであった。Tierra Version5.0では、このソフトは削除され、代わりに次節で述べるBeagleという観察ツールが用意されている。

3.10 Beagle

Beagleは初期バージョン[Kimezawa96]を木目沢が作成し、現在は同じ研究室の吉川氏が開発を担当している。このソフトもALmondと同様Tierraと独立したプログラムだが、機能がより強化されている。

スープメモリの状態の他に、「画面メニュー」の節で説明したほとんどすべてのメニューが使用可能である。またALmond Toolでは不可能だった、インターネット経由でのTierraの観察が可能である。

BeagleはTierra Verion5.0のソースコード(source.tar.gz)に添付されている。現在の所、GUIにX-windowを使用したUNIXマシンでのみ使用可能である。(LINUXでも可能。ただしコンパイルが必要。)

詳しくは吉川氏のホームページ(http://www.hip.atr.co.jp/~yosikawa)の情報をご覧いただきたい。

なお、ここで述べたBeagleはDOSバージョンのBeagle(現在はソースコードのdosviewというディレクトリにある)とは全く別のものなのでご注意いただきたい。

第 4 章

Tierraの応用

4.1 Tierra的手法による形態および適応行動の進化システム

この章では木目沢が行った、Tierraの類似研究について紹介する。4.2節では、Tierra的な命令語列で成長するルールが進化する樹木のシミュレーションについて紹介する。また、4.3節はTierra的な命令語列で、ゲームキャラクタの行動ルールを作成した例である。いずれもTierraのようにスープメモリ内でデジタル生物を進化させるのではなく、次図のように仮想空間内の個体（樹木、ゲームキャラクタ）毎に、成長や行動のルールをTierra的な命令語列で与え進化させようとした。そういう意味ではこれらのシステムはTierraと遺伝的アルゴリズムの中間的なシステムであるとも言える。

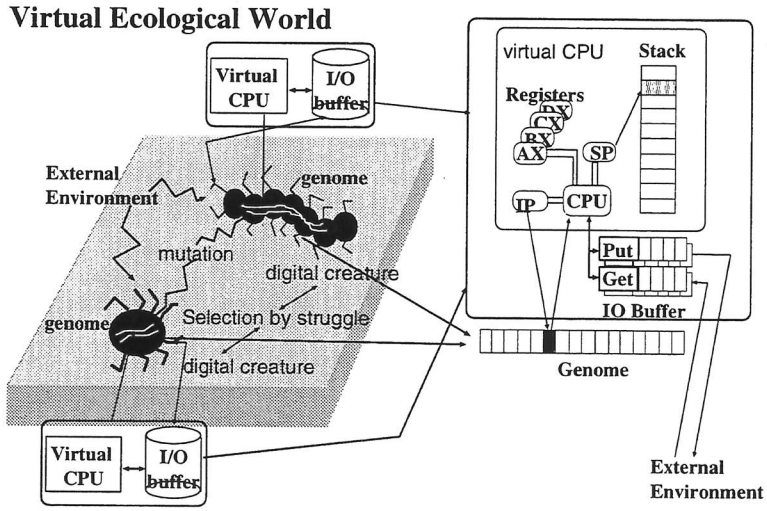


Figure 4.1: 仮想世界で進化する生物

この研究は、Tierraのような仮想機械語命令によるプログラムコードが少しでも実際的な問題に応用できるのかどうかを確かめようという試みの一つである。残念ながら4.3節の行動アルゴリズムの進化の方は実験ができないうちに中断してしまったが、4.2節の方は、ある程度の進化能力を持っている事が確認できた。

4.2 Tierra的手法による形態生成アルゴリズムの進化システム

体の基本的な形は遺伝子レベルで決っているであろうから、Tierra言語に体型を指示する命令語を組み込めば、これを突然変異させる事により様々な形態を進化させる事が可能である。ここでは仮想空間で機械語命令セットによるアルゴリズムを進化させる方法としてもっとも容易に実現できた、“仮想森”における樹木の進化について紹介する。

Tierraを形態生成に利用しようする場合、各仮想生物たちが競い合い、獲得するための対象はメモリとCPUではなく、体を作るために必要な3次元の空間とエネルギーとなる。例えば実際の植物では光合成を行うために、他の植物の陰にならないように、上へ上と成長していく。しかし、上へばかり成長しても日射を受ける葉の面積が広がらないため、効率がよくない。そこで、成長の初期に上へ伸びて後から葉を広げるか、先に横に広がって早期にエネルギーを蓄えてしまうかの二通りの戦略が考えられる。実際にはこれらの戦略を組み合わせて採用する事により、様々な形態の植物が進化してきたものと考えられる。

このシステムではTierraにおけるエネルギー資源のアナロジーのかわりに仮想的な日光を、物質・空間資源のアナロジーの代りに樹木を成長させるためのフィールド(土地)を与え、これらの資源をめぐって、より環境に適応した樹形を生成する個体が多く生き残るシステムであるようにした。

なおこの節の内容は[Kimezawa97]を元にしている。

4.2.1 個体モデル

仮想森における個体(仮想木)は、一つの遺伝コードと複数のセルによって構成された多細胞生物である。各セルは、それぞれ一つの仮想CPUと状態値(State)を持っており、状態値の違いにより、同一の遺伝コードの異なる部分を実行できる。よって各個体は全体としてみると、一つのプログラムの異なるコード領域を、複数のCPUで実行する**MIMD型並列計算機**とみなす事ができる。成長し子孫を残すために、各個体は最低一枚の“葉”と一つの“花”となる状態のセルを持たなければならない。

“葉”の状態のセルは、日射によるエネルギーを仮想CPUを駆動するためのCPU時間に変換する事ができる。従ってより多くの葉を日射を受けやすい位置に効果的に付けた個体ほど、たくさんの仮想機械語命令を実行できる。“花”の状態のセルは、特殊な仮想機械語命令“breed”を実行する事により、子孫(種)を自分の近傍に蒔く事ができる。種がまかれる範囲は、花の位置が高い程遠くまで広がる。またこのbreed命令実行時、各種の突然変異を生じる事がある。

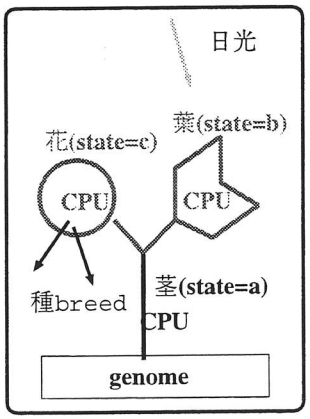


Figure 4.2: 個体モデル

4.2.2 成長の仕方(形態形成)

仮想木の成長はまず、形態記述用の仮想機械語命令により、**L-system**[Lindenmayer89]の文法(production rule)を各セルが持っている形態記述バッファに記述し、それを形態生成用の仮想機械語命令に翻訳する事により行われる。

L-systemはリンデンマイヤーが「発生のアルゴリズム」の概念をもとに考案したもので、様々な生物の形態の発生をうまくモデル化できる方法として有名である。例えば、発生過程に登場する細胞のタイプの集合をNとTする。ここで集合Nは書き換え可能なタイプ(非終端記号)、Tは書き換えできないタイプ(終端記号)である。またそれらの集合の要素で構成される文法(書き換え規則、production rule)Pを以下のように与える。

$$N = \{a, d, e, f, g, h, i, j, k, l, m, n, o, p, q, r, s, t, u, v, w, x, y, z\}$$

$$T = \{b, c, [,], \{, \}(,), <, >, +, -, !, \#\}$$

$$P = \{a \rightarrow a<b>\{c\}, \mathit{Other\ rules\ will\ be\ made\ by\ evolution}\}$$

このようなL-sytemにおいて、例えばFigure:4.3以下の図の左欄のような命令語列により、production ruleを生成させる。0(start命令)から5行目までの命令語列が中欄にあるような形態記述バッファにproduction ruleを記述する。次に6行目のstop命令でこのproduction ruleを3次元形状に翻訳し生成すると、右欄のような表現型が作られる。一つのプロダクションルールはstart命令ではじまり、stop命令で終了する。これは本物の遺伝子がたんぱく質を合成するときにおける、開始コドンと終止コドンに対応している。

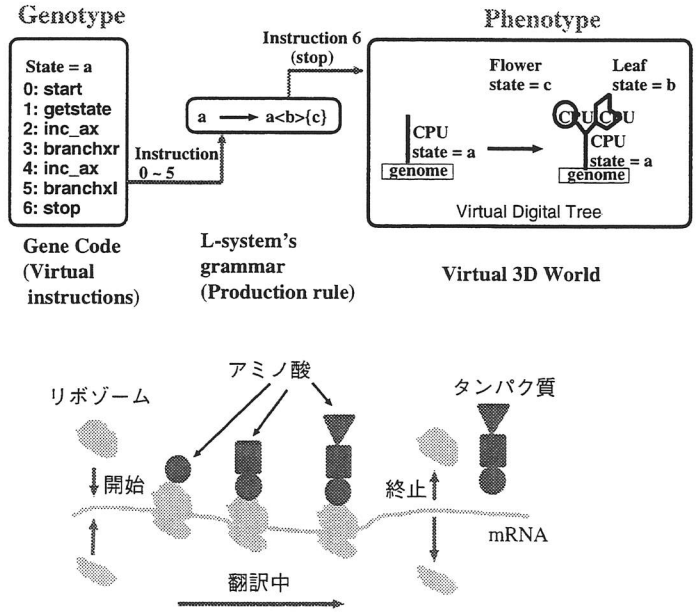


Figure 4.3: 成長の仕方(遺伝子型から表現型へ。mRNAからタンパク質への翻訳)

4.2.3 ゲノムの構造と仮想CPU

セルの状態によってゲノムの実行箇所を変えるために、一本のゲノムは一つまたはそれ以上の**染色体**に構造化されている。ここでは染色体という言葉を生物学本来の意味とは異なって用いている。また状態aのセルが実行するゲノムのブロックを染色体0番というように呼んでいる。

さらに各染色体は一つまたは複数の遺伝子に分けられる。遺伝子は“成長の仕方”の項でも説明したようにstart命令から始まり、stop命令で終る一連のコード列である。ただし、命令語の中にはstar、stop命令に囲まれていなくても有効なものもあるので、遺伝子の単位(区切り)は必ずしも明確ではない。

仮想森のシステムでは、まだ適応行動に関する機能は実現されていない。そのため、仮想CPUの構造は単純になっている。レジスタはAXレジスタひとつだけであり、これはほとんどの場合状態値の計算に使われる。IP(命令ポインタ)は実行すべき命令語のアドレスを指すが、その範囲は状態値に対応する染色体のアドレス領域に限定される。このCPUはproduction ruleを記述するために特別なバッファ**LSB**(L-System's Buffer)を持つ。**LSP**(L-System Pointer)はLSBへのプロダクション・ルールの記述先アドレスへのポインタである。またLSBへの書き込みはフラグ**mlfg**が真の場合のみ可能で、mflgはstart命令で真、stop命令で偽になるよう操作される。

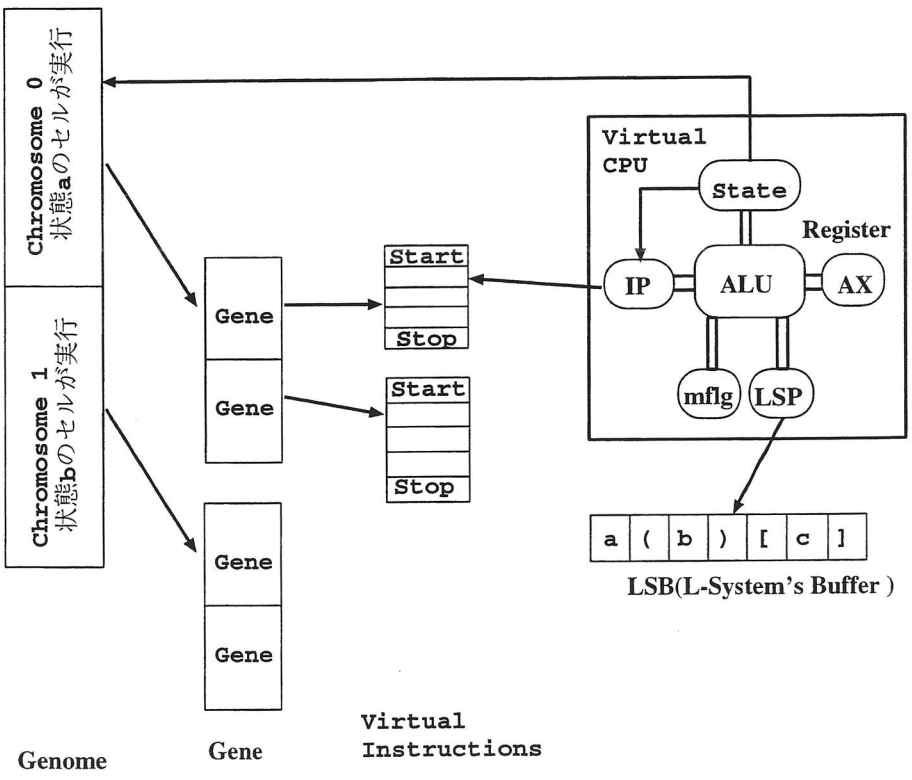


Figure 4.4: ゲノムの構造と仮想CPU

4.2.4 命令セット

次ページの表はこの研究で用いた命令セットの一覧である。“Morphorogical Synthesis” 行から始まる命令語が、形態記述・生成用の命令語群である。split,branchXX 命令は細胞分裂 (分岐) に関する記述を、inc_vangle ~ dec_hangle 命令で分岐角度の増減を、add_siz,dec_siz 命令で分裂セルのサイズの増減を行う。下図にこれらの命令語の例をいくつか示す。

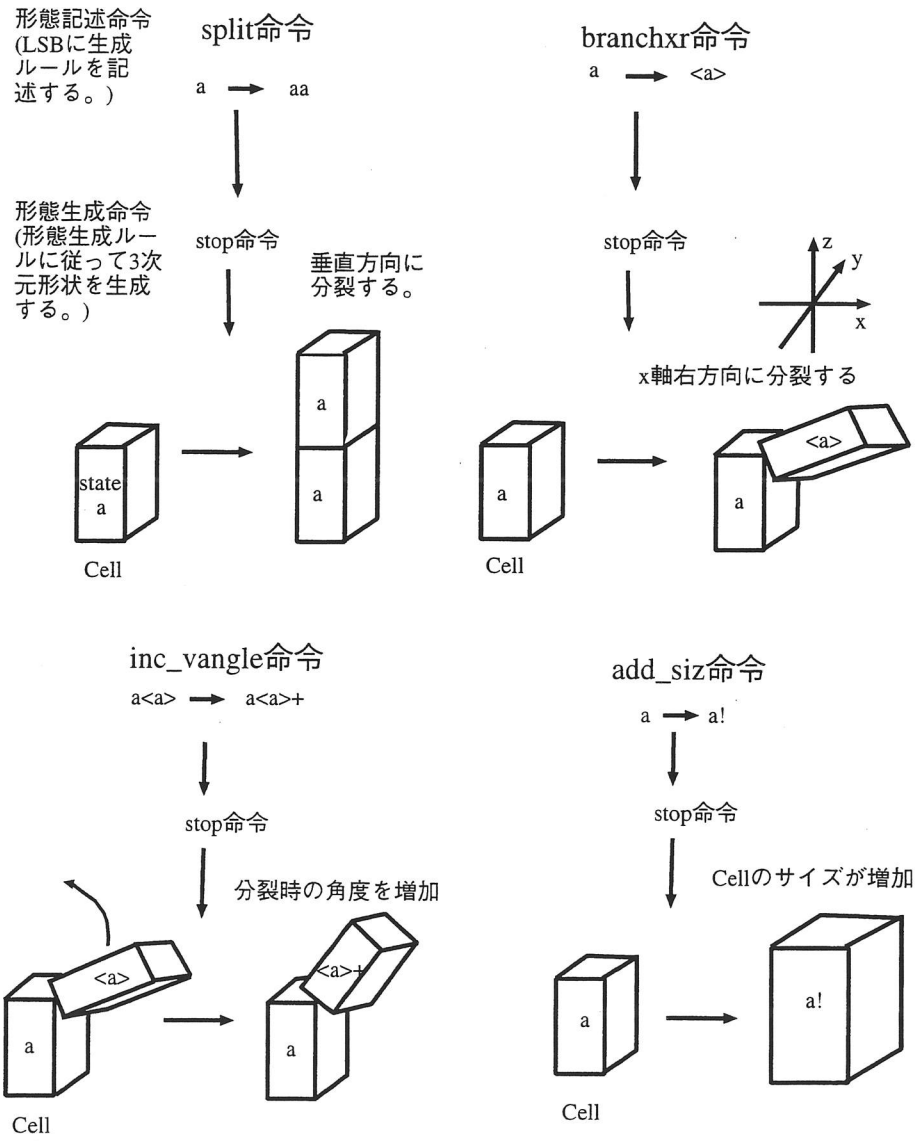


Figure 4.5: 形態記述・生成命令群の例。

ニーモニック	説明
No Operations: 2	
nop0	no-operaion, for template
nop1	no-operaion, for template
Calculation: 6	
zero_ax	set ax register to zero
inc_ax	increment ax, ax = ax + 1
dec_ax	decrement ax, ax = ax - 1
shl_ax	shift left ax, ax < < = 1
shr_ax	shift right ax, ax > > = 1
getstate	get the cell state and set, ax = "cell state"
IP manipulation: 6	
jmpf	move IP forward to template
jmpb	move IP backward to template
call	call a procedure
ret	return from a procedure
if_ax	if(ax==0) execute next instruction
halt	halt CPU
Morphorogical Synthesis: 14 (状態 b(leaf) と c(flower) の cell では無効)	
start	clear *LSB, set lsp = 1, set mflg = 1
putstate	if(mflg) then put ax+'a' to LSB[0], lsp ++
split	if(mflg) then put ax+'a' to LSB[lsp], lsp ++
branchxr	if(mflg) then put < ax+'a' > to LSB[lsp], lsp += 3
branchxl	if(mflg) then put {ax+'a'} to LSB[lsp], lsp += 3
branchyr	if(mflg) then put [ax+'a'] to LSB[lsp], lsp += 3
branchl	if(mflg) then put (ax+'a') to LSB[lsp], lsp += 3
inc_vangle	if(mflg) then put + to LSB[lsp], lsp ++
dec_vangle	if(mflg) then put - to LSB[lsp], lsp ++
inc_hangle	if(mflg) then put * to LSB[lsp], lsp ++
dec_hangle	if(mflg) then put / to LSB[lsp], lsp ++
add_siz	if(mflg) then put ! to LSB[lsp], lsp ++
dec_siz	if(mflg) then put # to LSB[lsp], lsp ++
stop	decode LSB and calculate geometrical parameter of generated cell. set mflg = 0
Biological: 1	
breed	breed a seed (State a cell) from the tip of cell. This is valid only for the state c(flower) cell.
Total: 29	

Table 4.1: 命令セット

4.2.5 仮想空間（フィールド・リーパー・エネルギー・物質・タイムスライス）

仮想木はフィールドとよばれる限られた土地に生まれ成長する（Figure **4.6**）参照。フィールドの全面積のうち仮想木が生育できる割合は限られており、ある設定面積（フィールド最大許容占有面積率）を越えるとリーパーと呼ばれる自然淘汰の機構が起動する。リーパーは各個体の評価値が少ない種ほど強い淘汰圧を受ける。ここで個体の評価値は以下の式で計算する。

評価値 = 1ステップ毎の個体全体の受光量/個体重量

シミュレーションではフィールド全面の垂直下方向に、単位面積当たりある設定された数の光子（単位日射量）を落とす。次にフィール内すべての個体の各細胞の3次元座標を計算しソーティング処理を行って、垂直方向に高い方から順番に光子を受ける。このとき葉の状態の細胞はその光子を命令語実行のためのエネルギーに変換する。よって個体重量に比較して受光量が多い（たとえば、葉の数が多いとか、葉の配置の仕方が効率的な）個体ほど、生存に有利となる。

ここでステップとは、日射を与えてから次の日射を与えるまでの間隔である。

エネルギーの値はそのステップにおけるタイムスライスとして、各個体に設定される。個体はそれを構成する細胞にそのタイムスライスを均等に与えて、命令語を実行させる。

なお、命令語のうち細胞分裂、細胞のサイズの変更、繁殖に関する命令語を実行するためには、その個体のある定められたエリア内に3次元形状を作るための物質がなければならない。この物質はシミュレーション開始時点では、全フィールドにある定められた量（単位物質量）で均等にばらまかれている。

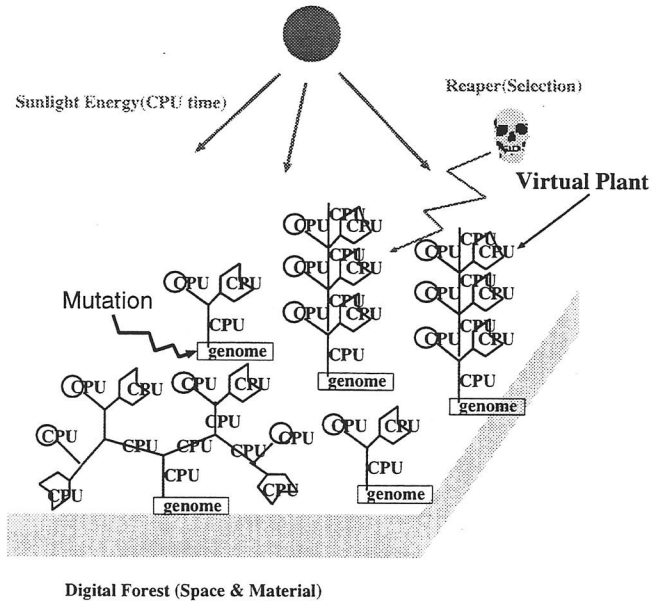


Figure 4.6: Digital Forest(仮想空間)

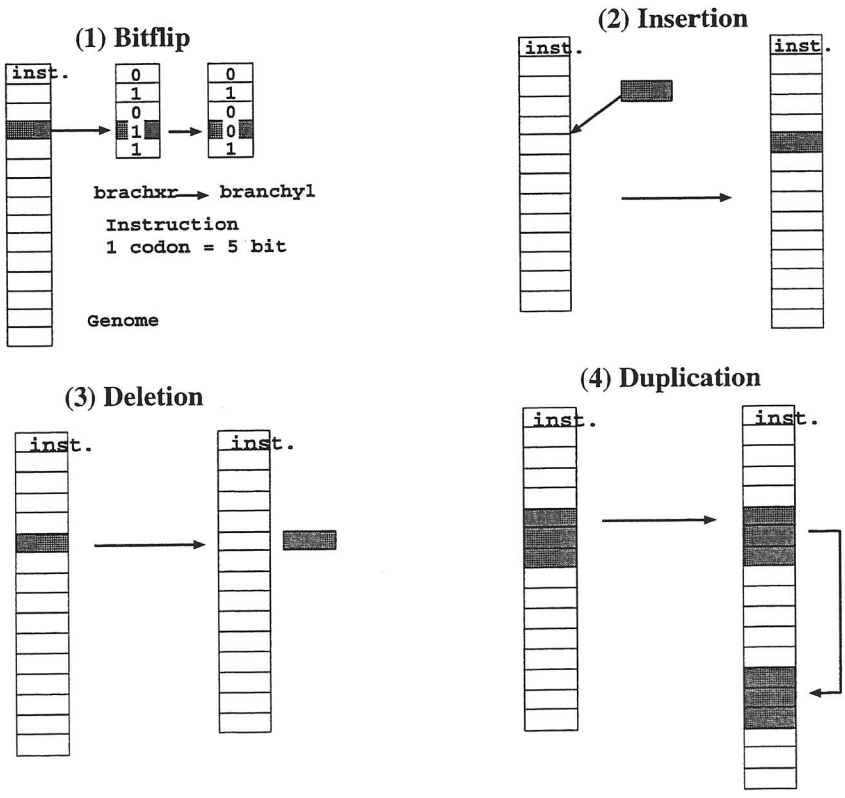
4.2.6 遺伝操作

遺伝子は環境設定されたある確率で突然変異する。本システムでは突然変異の種類として主に以降の図にある 9 つの方式を使っている。

(1) から (4) は命令語レベルの変異である。すなわち命令語を構成するビットの反転 (Bitflip(1))、命令語と命令語の間にある別の命令語が挿入される Insertion(2)、その逆にある命令語が削除される欠失 (Deletion(3))、ある命令語列が別の場所にコピーされる重複 (Duplication(4)) を用意した。とくに (4) 重複は遺伝子の元の機能を残したまま、別の機能を獲得させる事ができるようになる準備段階として非常に重要な変異である。

(5),(6) は染色体レベルの変異である。染色体がまるごと削除されたり、重複したりする変異を用意した。これにより染色体レベルの機能の増加が可能なようにした。すなわち、ある細胞が今まで持っていなかった状態 (State) を持つように遺伝子が変異したとき、その状態に対応する染色体を持っていなければ、その細胞は機能を停止してしまう。しかし、染色体レベルの重複によりゲノムは機能できる状態数を増加させる事ができる。

(7) ～ (9) は遺伝子レベルの変異である。当初、これらの変異は用意していなかった。しかし、本システムでは、遺伝子レベル (start 命令から stop 命令までで構成される命令語単位) で、一つの単位となる機能を実現している。したがって、機能単位レベルの進化を加速させるためにこれらの変異を導入した。遺伝子レベルの Insersion(7),Duplication(8) そして交叉 (Cross over(9)) を用意した。



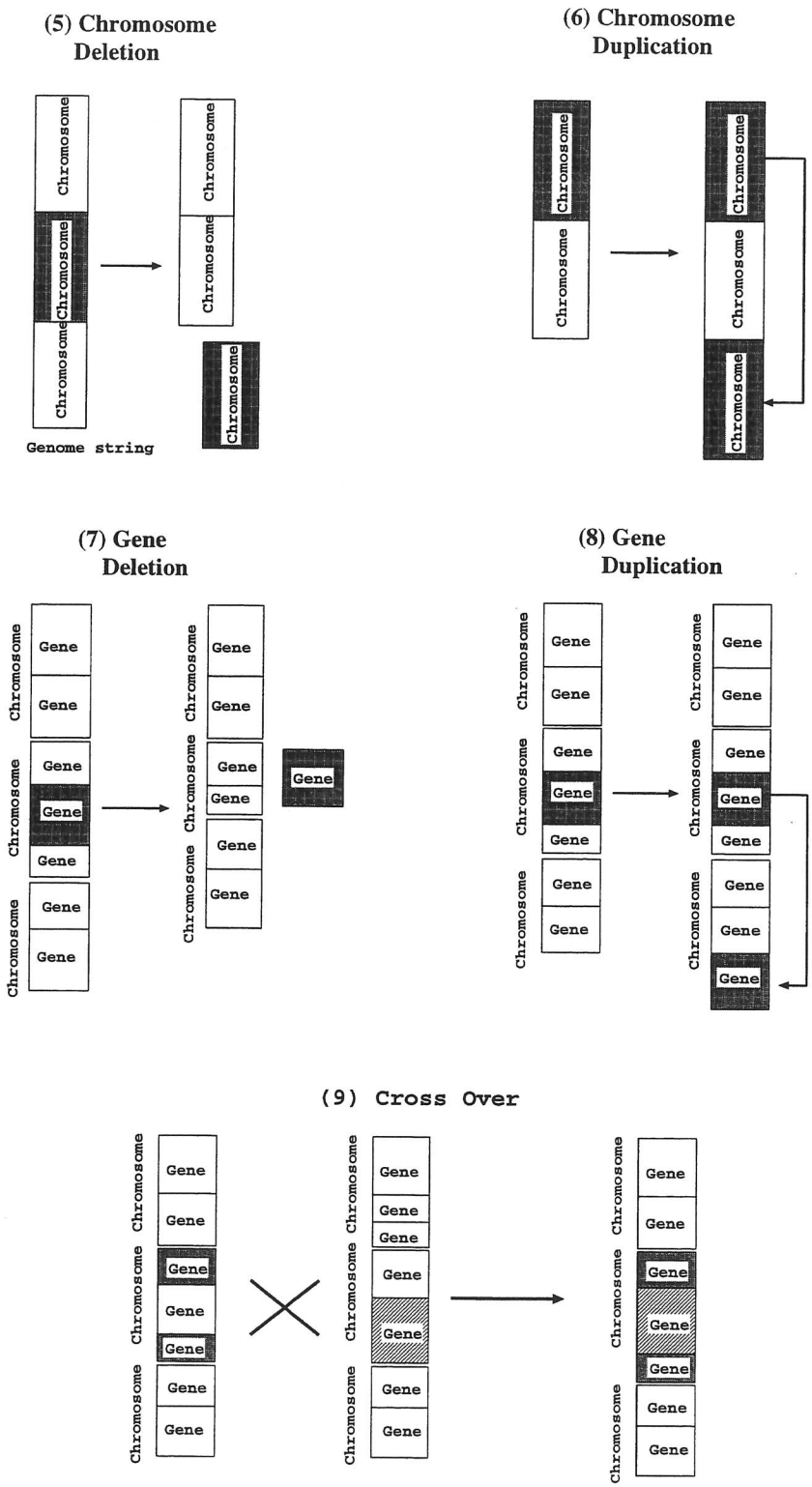
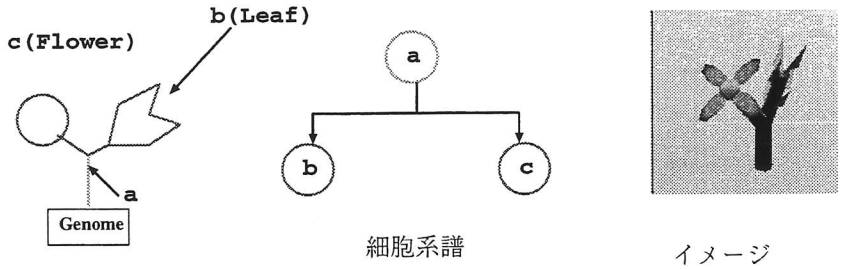


Figure 4.7: 遺伝操作

4.2.7 先祖種

先祖種は子孫を残せる最もシンプルな構造とした。ゲノムは１３個の仮想機械語命令からなる。またこのゲノムは3つの染色体にわかれており、初期状態(状態値a)のセルは染色体0(0から7行目までの命令語列)を実行する。初期状態のセルは細胞分裂により、状態値b(葉)と状態値c(花)のセルを作ったあと、halt命令によりCPUの実行を停止する。葉状態のセルは染色体1のコードを実行するが、この染色体にはnop0命令しかないので何もしない。日射を受けてCPU時間を得る事に専念する。花状態のセルは染色体2のコードを実行する。breed命令で種をまいた後、jmpb命令でこの染色体の先頭アドレスに戻り、breed命令を繰り返す。なお以下の図の真ん中に**細胞系譜**を、右側に3Dイメージを示す。



細胞系譜　　イメージ

Addr.	命令語	説明
chromosome 0 (state a)		
0	start	Clear LSB, lsp=1, mflg=1
1	zero_ax	ax = 0
2	inc_ax	ax = 1
3	branchxr	LSB = a < b >, lsp=4
4	inx_ax	ax = 2
5	branchxl	LSB = a < b > {c}, lsp=7
6	stop	decode LSB and generate 3D shape, mflg=0
7	halt	halt this cell's cpu
Chromosome 1 (state b)		
8	nop0	no-operation
Chromosome 2 (state c)		
9	nop0	start address of chromosome 2
10	breed	breed (種をまく)
11	jmpb	jump backward to start address
12	nop1	complementary template of start address of chromosome2

Table 4.2: 先祖種とそのコード

4.2.8 計算例

シミュレーションを行うには、命令セットの設計、先祖種の設計、仮想空間のパラメータの設定(フィールドサイズ、日射の強さ、突然変異率など)が必要である。

項目	値
フィールドパラメータ	
フィールドサイズ	36.0 × 36.0 (端と端はつながっている。ドーナツ状。)
単位日射量	光子2個 / 面積 / ステップ
単位物質量	2個 / 面積
個体パラメータ	
オリジナル細胞サイズ	0.5 × 0.5 × 1.0 (直方体モデル)
取りうる最大状態数	26 (a ～ z)
最長ゲノムサイズ	9999
最大成長サイズ	末端の枝から根本まで6個以下の細胞で構成される事。
命令語特有のパラメータ	
branch 命令群による垂直方向への分岐角度	45 度
inc_hangle による水平方向への分岐角度増加量	15 度
dec_hangle による水平方向への分岐角度減少量	15 度
inc_vangle による垂直方向への分岐角度増加量	15 度
dec_vangle による垂直方向への分岐角度減少量	15 度
add_siz による細胞サイズの増加量	20%
dec_siz による細胞サイズの減少量	20%
Mutation Rate	
Bitflip	0.20
Insertion	0.025
Deletion	0.025
Duplication	0.025
Gene insertion	0.05
Gene deletion	0.05
Gene duplication	0.05
Chromosome deletion	0.025
Chromosome duplication	0.025
Chromosome cross over	0.05
Total mutation rate	0.50

Table 4.3: 実験パラメーター

4.2.9 Overview image

シミュレーションのイメージの一例を口絵 4 に示す。なお細胞モデルの形状は計算上は直方体であるが、本イメージではより植物らしく見せるため円柱で表している。とくに状態 b と状態 c の細胞には、それぞれ花と葉をポリゴンで作成し表示している。

変異した種の花びらの色は親の花の色に比べある程度ランダムに変化するようにした。さらにゲノムが長い程、より花びらの枚数がより多くなるようにした。また、花その物の色（花びらの内側の球で表示）は、その個体が取りうる状態数を表すようにしている。先祖種（状態数=3）の花の色は緑だが、状態数が増える毎に色がかわる。

なお本シミュレーションでは、計算機には Sun Sparc Ultral1 Creator3D (167MHz) を使用し、コンピュータグラフィックスの描画には OpenGL1.1 を利用した。なお、SGI の IRIX OS 搭載マシンでも実行可能である。

- ステージ中央に先祖種（状態 a) が 1 体だけ、ある位置から成長を始める。(0 Step)
- 先祖種は花に成長し繁殖する。(16 Step)
- 先祖種や少し変異した種がフィールド一杯に繁殖する。(28 Step)
- 先祖種より細胞数が多く背の高い種が現れる。(52 Step)
- 背の高い種がフィールドのほとんどを占める。(2016 Step)
- さらに枝別れ（成長の仕方）が複雑になった種が現れる。(12096 Step)

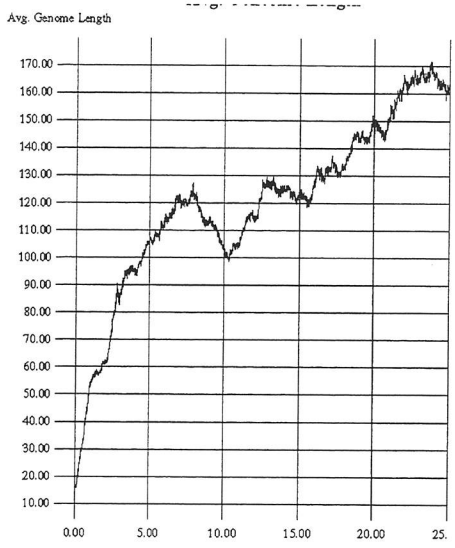
4.2.10 ゲノムサイズの増加とパフォーマンスの向上

次ページにこのシミュレーションでの平均ゲノムサイズ (1) と主な種のゲノムサイズの分布 (2)、生物が獲得した全タイムスライス（エネルギー量）(3)、平均評価値 (4) をステップ毎に表したグラフを示す。

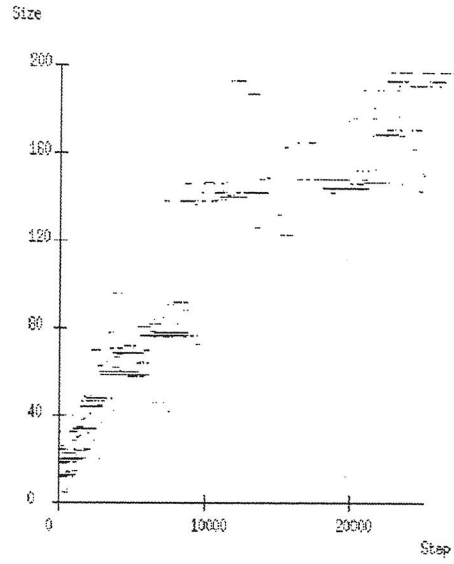
各グラフとも右方上がりで上昇している。平均ゲノムサイズは先祖種の 13 から 25000 ステップでは約 170 までに増加した。(2) の主なゲノムサイズのグラフは、フィールド上の全個体数のうちある設定した割合（10%）を越えた種のゲノムを表示したものである。進化が断続的に進行して行った事がうかがわれる。

(3)(4) のグラフをみると、約 10000 ステップ目位いで急に全タイムスライスと評価値が向上した事がわかる。これはたぶん口絵 4 の 2016 ステップのイメージ図で見られるような背の高い生物から 12096 ステップのイメージ図で見られるより枝別れ構造が複雑化した生物への進化が起こったためと思われる。

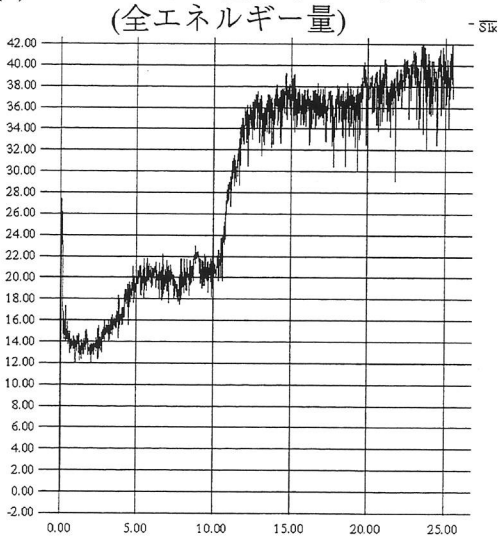
(1) 平均ゲノムサイズ



(2) 主な種のゲノムサイズ



(3) フィールドの全タイムスライス (全エネルギー量)



(4) 平均評価値

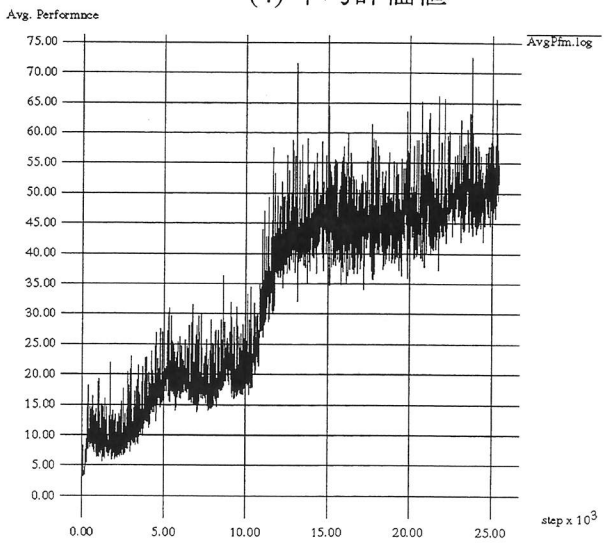


Figure 4.8: シミュレーション結果

4.2.11 進化

シミュレーションの途中で現れた種をいくつか紹介する。本研究では、新しい遺伝子型を持った種が現れ、その繁殖個体数があるしきい値を越えると、ID 番号とともに遺伝子バンクに登録されるようになっている。ID は、

ゲノムサイズ (4 桁)- 生まれたときのシミュレータの全命令実行回数（単位百万命令）

によって付与される。

ID0013-0.054275

ゲノムは先祖種とほとんど同じだが、染色体 0 の 2 番目と 3 番目の命令語の間に split 命令が挿入され、さらに 8 番目の命令語 halt が無くなる事によって、繰り返し茎を成長させて、背が伸びる種である。評価値は 3.30 と先祖種の 4.45 より低いが背を伸ばす事により日射を受けるのに有利であり、花の位置が高いので種を広い範囲にまく事ができる。以下にゲノム、3D イメージ、細胞系譜を示す。

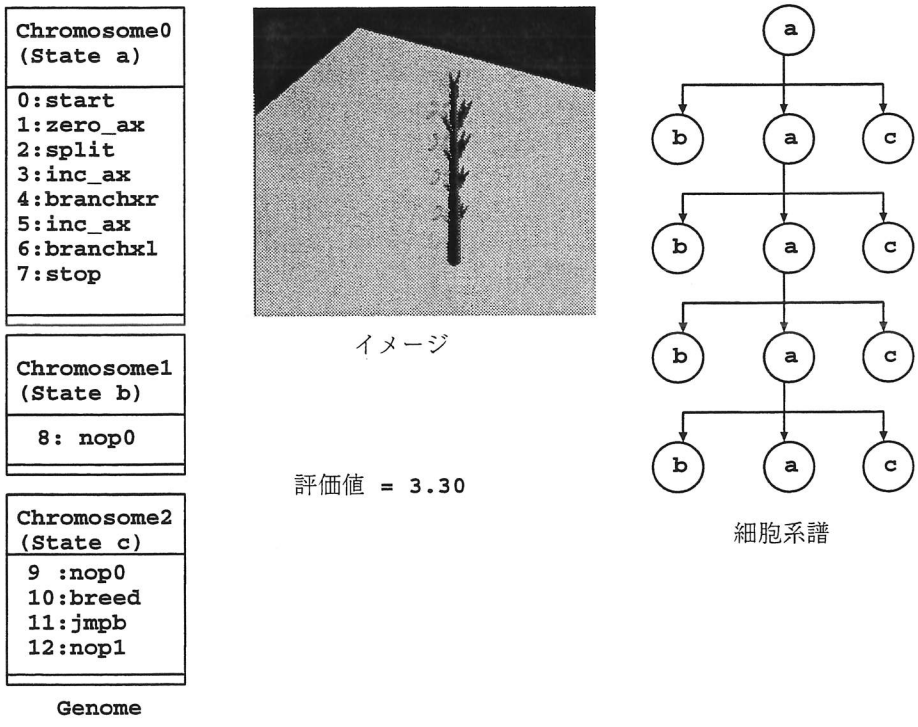


Figure 4.9: ID0013-0.054275

ID0085-11.059977

形がいいので紹介する。評価値も 30.26 とまあまあである。

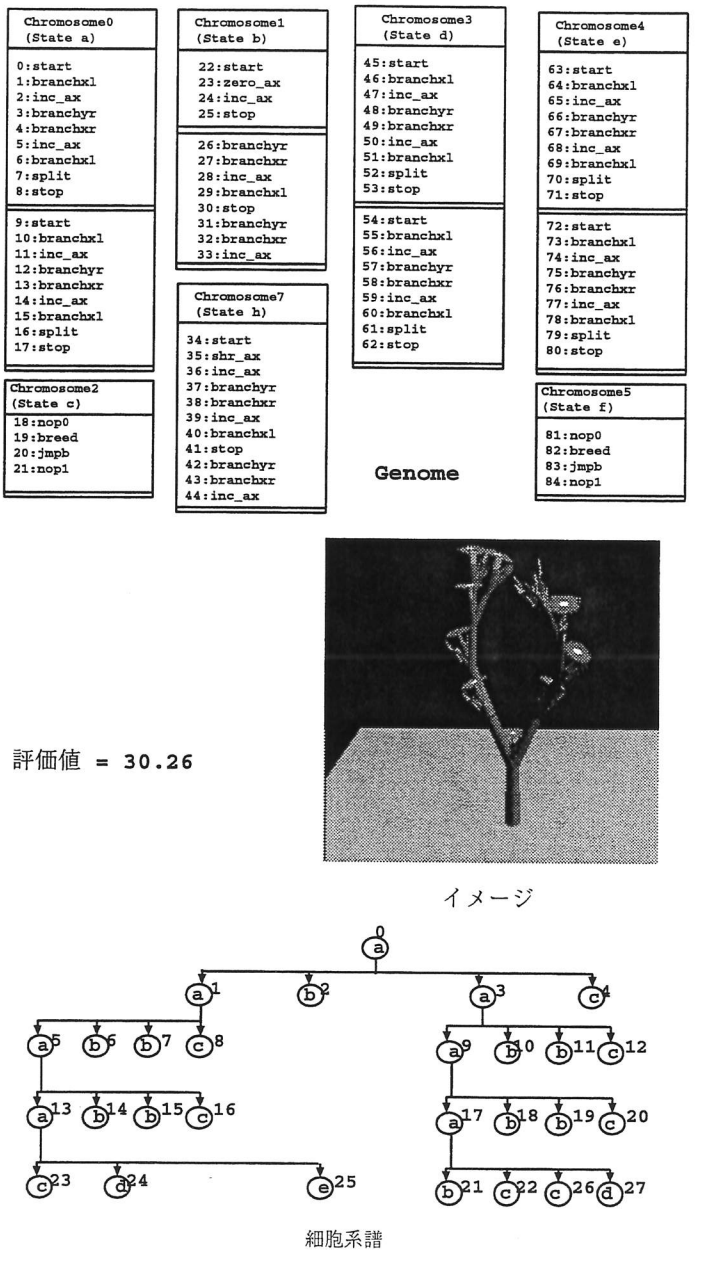


Figure 4.10: ID0085-11.059977

ID0137-141.044853

kのシミュレーションの後期に出現した。染色体の重複により状態数が12個に増え、枝別れ構造がかなり進化している。ただしゲノムを解析してみると染色体3,5,7,8は使われていない。イントロンのような状態になっている。評価値は55.8とかなり向上している。

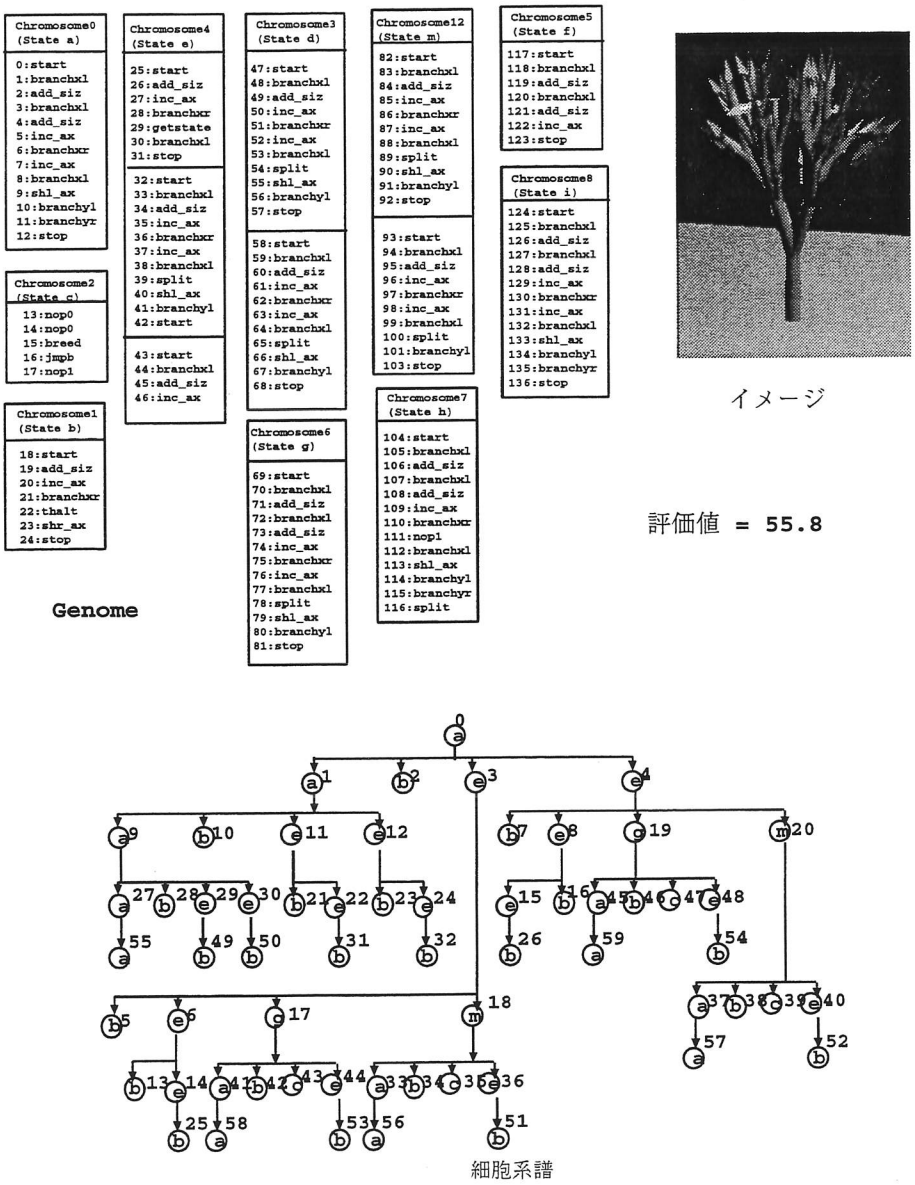


Figure 4.11: ID141.044853

4.2.12 考察と課題

本研究ではTierra的な方法によって植物の形態形成アルゴリズムを進化させる方法についてシミュレーションを行った。その結果、ゲノムサイズを増加させ、それと共に形態を複雑に進化させる事ができた。

Tierraでは限られたメモリスープの中で自己複製する種を進化させている。そのためゲノムサイズ自身が表現型の一部となる。その結果、サイズを増加させながらアルゴリズムを複雑化させるのが困難である。本研究ではゲノムをメモリスープから解放したため、そのような制約はなく、重複などの突然変異によって、ゲノムサイズを増大させる事ができる。その反面、Tierraのような命令語同士のダイナミックな相互作用がないため、進化して得られるアルゴリズム自体は単純である（テンプレートを利用した寄生・免疫関係などは起こらない）。

また、本研究では生物の形としては比較的単純な植物の成長の仕方をL-systemと組み合わせて進化させた。L-systemにはさまざまな拡張があり、もっと複雑な植物の形態や動物の形態生成に使われている場合もある[Doi63]。よってL-systemの生成ルールを拡張する事により、さらに複雑な形態を生成させる事も可能だろう。その場合問題になるのは評価値をどのように設定するかである。受光量/重量という単純な評価値ではうまく行かないのはあきらかである。そこで、仮想森の世界に昆虫などの動物を加えて、それらとの共生関係により、花の色や形を進化させるなどの方法が必要であろう。

このシステムでは仮想CPUの機能もかなり限定され、レジスタは一個だけ、I/O関係の命令もなかった。そこで日射の向きを時刻によって変わるようにし、その方向を得るセンサーの機能を果たす命令語を持った細胞を設計すること（光の向き強度を入力し、その値を出力して他の細胞にわたす）により世界との相互作用が生物ができるかもしれない。

4.3 Tierra的方法による行動アルゴリズムの実現法

Tierra的な方法で行動アルゴリズムを生成するのは、形態生成アルゴリズムよりはるかに難しい。2次元格子のような単純な仮想空間を仮定し、その中で単純な移動命令や攻撃命令を仮想機械語命令として実装した例[坂無94]があるが、それ以上の複雑な世界（我々が住んでいるような現実の世界）で行動するアルゴリズムを実現するのは、極めて困難な課題である。

そこでここでは、機械学習のアルゴリズムを進化的に生成させる方法として最もよく用いられている分類子システム[Goldberg89]と仮想機械語命令の組合せにより、仮想3次元空間で行動するゲームキャラクタの行動アルゴリズムを実現させようとした例について紹介する。

4.3.1 ゲームキャラクタの知能システム

ここでは、ある3Dゲームに登場するゲームキャラクタの行動アルゴリズムについて説明する。仮定しているゲームは、プレーヤーが操作するプレーヤー・キャラクタと人工知能によって自律的に行動するゲームキャラクタの2種類が用意される。人工知能は

分類子システムと仮想機械語命令を組み合わせたシステムとして実現する。ゲームキャラクタの初期行動として、3D空間に落ちているアイテムを探し回りそれらを拾ってプレーヤーに届ける行動を実装した。

分類子システムにもいろいろな種類があるが、ここでは最も単純な方法であるCognitive System 1(CS-1)[Holland78]を参考にした。

ゲームキャラクタは、「アイテムを探す」、「アイテムを拾う」、「プレーヤーを探す」、「仲間を探す」、「餌をもらう」、「餌を食べる」、「歩き回る」、「休む」、「攻撃する」、その他の基本的行動を行うものとする。

基本行動そのものは分類子システムで選択する。例えば、お腹が減っていて、アイテムをすでに持っているなら、餌をもらいにプレーヤーの所へ行くなどのように、ある条件が成立すると、ある行動を行うという形式で行動を選択する。

そして、これらの基本行動の内容を仮想機械語でより具体的に実装する。たとえば「アイテムを探す」ならば、どこまでの範囲をどのように探すのか、「歩き回る」なら、どんな速度でどの方向へ歩くのかといったより細かな行動単位を実装する。

4.3.2 分類子システム(Classifier System)

今回用いた分類子システムの構成をFigure:4.3.2に示す。このシステムには2つの分類子がある。ひとつは内部状態(活力の有無、アイテムの有無)に関するもので、センサーマクロを選択する。これを**内部状態・センサークラシファイアー**と呼ぶ。もう一つは、センサーマクロとその実行結果(探した対象が見つかったか見つからなかったか)と比較処理される分類子で、キャラクタの行動を選択するためのものでありセンサー・アクションクラシファイアーと呼ぶことにする。

まず、キャラクタは内部状態を**内部状態・センサークラシファイアー**と比較する。classifireを構成する遺伝子と内部状態が一致(matching)した場合、その遺伝子のセンサーマクロが選択実行される。もし、複数の遺伝子とmatchingした場合は重み(weight)によって確率的に選択される。また、matchingする遺伝子がない場合は乱数により遺伝子を選択する。

続いてセンサーとその実行結果をセンサー・アクションクラシファイアーと比較し、アクションマクロを選択する。この場合は内部状態・センサークラシファイアーと同様に重みにより確率的に遺伝子を選択する。

これら一連の遺伝子の選択結果は一時的に記憶される。そしてアクション実行の結果、もし特別なイベントが起こった場合(アイテムを拾った、食べ物を食べた、敵に攻撃されたなど)に、報酬としてそれら遺伝子のweightを調整する。よってアイテムを拾った、食べ物を拾ったなどの良いイベントが起こった場合はweightは大きくなり、敵に攻撃されたなどの悪いイベントが起こった場合はweightは小さくなる。

また、アクションが実行される毎に内部状態は修正される。例えば移動すれば、活力はやや減少する。活力は最大値が1.0、最小値が0.0であるが、分類子と比較される場合、0.5以上なら1、0.5未満なら0に量子化する。

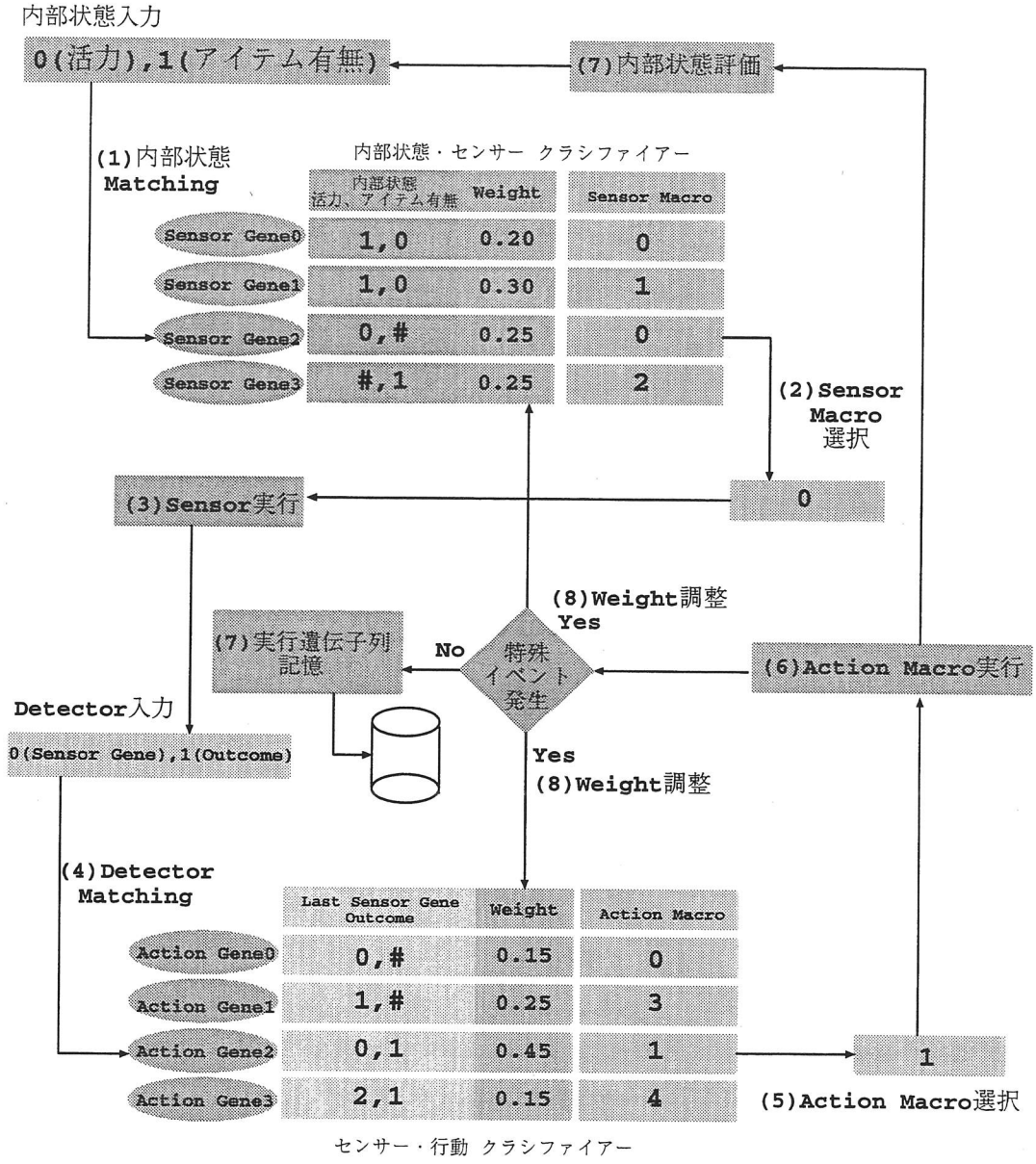


Figure 4.12: Classifire System

4.3.3 仮想機械語命令による行動マクロ

行動マクロは仮想機械語命令で構成される。この知能システムはもともと、オンライン・ネットワークゲームの知能システムを意識して作成された。そしてclassfire systemと仮想機械語による知能モジュールを一般公開し、ゲーム参加者自身にも知能モジュールの作成を行ってもらえるようにと考えていた。そのため、一旦知能モジュールの仕様を公開してしまうと、その後の変更は非常に難しくなる。またゲームの具体的内容も未確定な所が多かったので、どのような行動マクロや仮想機械語命令を用意しておけばよいかも不明確であった。

そこで、命令セットはなるべく汎用的に使用できるよう考慮した。すなわち、「アイテムを探す」、「餌を食べる」などのような具体的対象物に対して単機能的に使う命令語にではなく、no-operation命令との組合せによりいろいろな意味にかわるような命令セットを考案した。

ニーモニック	説明
No Operations: 2	
nop0	no-operaion, for template
nop1	no-operaion, for template
Calculation: 7	
zero_ax	set ax register to zero
inc_ax	increment ax, ax = ax + 1
dec_ax	decrement ax, ax = ax - 1
shl_ax	shift left ax, ax << = 1
shr_ax	shift right ax, ax >> = 1
lda	load a data and set it into AX from data area (adressed by template).
sta	store a date from AX onto data area (adress by template).
IP manipulation: 6	
jmpf	move IP forward to template
jmpb	move IP backward to template
call	call a procedure
ret	return from a procedure
if_ax	if(ax==0) execute next instruction
Behaviorla command: 3	
search	search an object. (Object is specified by template).Search range is specified by the value in AX.
maction	move action. (action type is specified by template). Move speed is specified by the value in AX.
saction	special action. (action type is specified by template).
Total: 18	

Table 4.4: 命令セット

例えば、同じsearchという命令語に対して、その直後にnop0が続くかnop1が続くかによって、何を探すのか意味が変わるようにした。search,nop1,nop0なら、プレーヤーを探すという意味であり、search,nop0ならアイテムを探すという意味になるように設定した。またmactionは移動を表す命令(move action)とするが、maction,nop0なら直進、maction,nop1なら右へ方向転換などのようにする。さらにsaction(special action)命令の場合は、saction,nop0ならアイテムを拾う。saction,nop1ならアイテムを渡すなどのようにした。

これら、search,maction,sactionの後に続くno-opration命令の数はその意味がシステム側であらかじめ設定されていればいくつあってもかまわない。

こうする事によって、行動マクロのステップ数は増えるが少ない命令語で拡張性のある命令セットを用意する事ができた。

またある行動をする場合、例えば一口に前進すると言っても、どの位の速さで前進するのかなどの細かいパラメータも指定できるとよい。そのために、本システムでは行動マクロを命令語領域とデータ領域に区別し、データ領域にパラメータとなるデータをセットできるようにした。データ領域からデータを読み出す場合は、lda命令などを使う。ldaはデータ領域のデータをAXレジスタにセットせよという命令語である。データ領域のデータのアドレスはno-operation命令によるテンプレートによって指定する。テ

ンプレートがnop1,nop0ならば、これは二進数の10すなわち十進数の2を表すのでデータ領域のアドレス2にあるデータを指す事になる。

4.3.4 基本行動の例(アイテムを探してプレーヤーに届ける)

前ページの命令セットを用いて、先祖種の基本行動を作成した。基本行動は、センサーマクロ群と行動マクロ群の二つに大別される。センサーマクロ群はアイテムを探すマクロと、プレーヤーを探すマクロからなる。行動マクロは休息、移動、方向転換、アイテムを取る、アイテムを渡す行動マクロからなる。

Macro, Address	ニーモニック、説明	
Sensor Macro 0	範囲100m以内のアイテムを探す	
Inst[0]	lda	AXに100.0をセット
Inst[1]	nop0	データアドレス0
Inst[2]	search	searh,nop0はアイテムを探す命令
Inst[3]	nop1	
Data[0]	100.0	
Sensor Macro 1	範囲1m以内のアイテムを探す	
Inst[0]	lda	AXに1.0をセット
Inst[1]	nop0	データアドレス0
Inst[2]	search	search, nop0はアイテムを探す命令
Inst[3]	nop0	
Data[0]	1.0	
Sensor Macro 2	範囲100m以内のプレーヤーを探す	
Inst[0]	lda	AXに100.0をセット
Inst[1]	nop0	データアドレス0
Inst[2]	search	search,nop1,nop0はプレーヤーを探す命令
Inst[3]	nop1	
Inst[4]	nop0	
Data[0]	100.0	
Sensor Macro 4	範囲1m以内のプレーヤーを探す	
Inst[0]	lda	AXに1.0をセット
nst[1]	nop0	データアドレス0
Inst[2]	search	search,nop1,nop0はプレーヤーを探す命令
Inst[3]	nop1	
Inst[4]	nop0	
Data[0]	1.0	

Table 4.5: アイテムを探してプレーヤーに届けるセンサーマクロ

4.3.5 進化

以上のように本システムは分類子システムと仮想機械語命令により構成される行動マクロとのハイブリット・システムである。したがって、本システムに突然変異などの進化的手法を取り入れる場合にも、分類子システムの遺伝子を変異する方法と、行動マクロを構成する仮想機械語命令列を変異させる方法の2つのレベルがある。実際には両方を変異させる事になるだろう。その場合、探索空間は2つのレベルのかけ算となるため極めて膨大なものになる。通常分類子システムによる探索空間だけでも巨大なものになるが、このシステムでは分類システムで選択されたセンサー、行動マクロの詳細自身も進化可能であり、通常の進化・学習システムで扱えるレベルを越えている。

Macro, Address	ニーモニック、説明	
Action Macro 0	休む	
Inst[0]	maction	maction,nop0は行動市内（休憩）を意味する。
Inst[1]	nop0	
Action Macro 1	前方に移動	
Inst[0]	lda	AXに50.0をセット
Inst[1]	nop0	アドレス0
Inst[2]	maction	maction,nop1は前方に移動を意味する。
Inst[3]	nop1	
Data[0]	50.0	最高速度の50パーセント
Action Macro 2	方向転換	
Inst[0]	lda	AXに15.0をセット
Inst[1]	nop0	アドレス0
Inst[2]	maction	maction,nop1,nop1は方向転換を意味する。
Inst[3]	nop1	
Inst[4]	nop1	
Data[0]	15.0	回転角度（度）
Action Macro 3	アイテムを取る	
Inst[0]	saction	saction,nop0はアイテムを取る事を意味する。
Inst[1]	nop0	
Action Macro 4	アイテムを渡す	
Inst[0]	saction	saction,nop1はアイテムを渡す事を意味する。
Inst[1]	nop1	

Table 4.6: アイテムを探してプレーヤーに届ける基本行動マクロ

また現在は内部状態として、活力とアイテムの有無しか設定していないが、喜怒哀楽などの感情や自分の位置の認識（家の内外など）を追加する事により、より複雑な状況に対応できるようになる。さらに行動の種類も敵に対する攻撃や防御など追加しなければならない。だがそうするとますます探索空間が広がってしまう。

そのため、コンピュータによる突然変異だけでなく、ゲームを遊ぶユーザー自身がこの知能ルーチンの作成に積極的に介入して、より適応的なゲームキャラクタに育て上げられるようにする方法を取り入れるのが現実的であろう。そのためには、ユーザーにとって使い勝ってのよいユーザーインターフェーズや、たくさんのユーザー間でのデータ交換などゲームとしてのインフラを整備する必要がある。

残念ながら本システムはゲーム・システム本体の構築と同時平行的に作業が行われていたため、まだ進化実験は行われていない。本システムの有効性と改良については今後の研究に委ねられる。

第 5 章

おわりに

Tierraは未だ研究途上であり、将来どんな成果を生み出すかまだまだわからない。とくにNetwork Tierraに関しては、研究開始後5年を過ぎたがまだ最終の研究結果は出ていない。そういう意味でここでは結論めいたことを言う事はできないが、私の個人的な感想を述べたいと思う。

私がTierraを知ったのは、ここATR人間情報通信研究所に派遣されてからの事である。1993年4月に納入された超並列コンピュータCM-5のシステム管理者として私は赴任した。ここの第6研究室は下原勝憲室長を中心として人工生命研究の中心的な研究所を作るべく発足された。その年の8月トム・レイ博士が迎えられ、本格的な研究が開始された。私はその翌年からTierraプロジェクトに参加し、Network TierraのTCP/IPプログラムや、観察ツールBeagleの開発に携わった。Tierraを見てまず感じたのは、一見でたらめのようで実は極めて精巧なアルゴリズムが次ぎ次ぎに生まれてくる事の不思議さであった。これは多くの人工生命システムに共通した特徴だが、秩序と無秩序の境界で起こるこのような現象が、ソフトウエアという精巧さを要求されるシステムでも起こる事を目の当たりに見せてくれた。そして進化というのは人間が作った仮説ではなく、自然な現象である事を実証してくれた。

ところで、Tierraは一体何の役にたつのですかという質問をよく受ける。それに対してトム・レイ博士は複雑な並列プログラムを自動生成したり、将来ヒトの役にたつ知的エージェントの研究に利用できるなどのような事をおっしゃっているが、たぶん本人自身それは重要な事ではないと思っているだろう。

私の個人的な意見では別に何の役に立たなくてもいいと思う。なぜならTierraは生命の本質とは何かという科学にとって最も大きな疑問に答える有効な道具の一つであり、その事だけで充分役にたっていると思うからだ。

5.1 関連情報

Tierraに関するさらに詳しい情報は、トム・レイ本人のホームページ、

http://www.hip.atr.co.jp/~ray/tierra/tierra.html

が一番豊富である。

また、サンタフェ研究所のSimon Fraser氏は“MacTierra”というMacOSで動くTierraを独自に開発した。URLは以下の通りである。

http://www.santafe.edu/~smfr/mactierra.html

Tierraに類似した独自のシステムで注目すべき研究が、私の知るところ2つある。

一つはカルフォルニア工科大学(UCLA)のChris Adami氏らを中心として行われている、“Avida”(an auto-adaptive genetic system)である。URLは以下の通りである。

http://www.krl.caltech.edu/avida/

もう一つはATR人間情報通信研究所の鈴木秀明氏が行っている“Semar”(the sea of matter)である。氏のホームページは以下の通り。

http://www.hip.atr.co.jp/~hsuzuki/

Tierra関係の研究に興味のある方はこれらの研究についても、注目していただきたい。

5.2 謝辞

トム・レイ博士は昨年(1998年)、米国オクラホマ大学の動物学部の教授に赴任されました。ATR在任中は公私にわたりご指導くださり、大変感謝致しております。今後も年に1,2回はATRに来訪されるとのことです。

現在ATRにおけるTierraの研究は同じ第6研究室のJoseph Hart氏と、吉川徹氏が引き続き行っています。両名にはBeagleの開発引き継ぎなどで大変お世話になりました。

また本レポートの第4章Tierra応用は、同研究室の和田健之介氏にも研究指導いただきました。ここにお礼申し上げます。

本レポートの著者本人は、この春(1999年3月)をもってATRでの勤務を終える事になりました。ATRでお世話になった多くの皆様、とくに下原勝憲室長をはじめとする第6研究室の方々に深く感謝いたします。

付録 A

用語集

この用語集を作成するにあたっては、

「岩波生物学事典」[Iwanami],
「進化生物学」[Futuyma],
「生物のかたちづくり」[Doi63],
「エージェントアプローチ人工知能」[Stuart],
「適応エージェント」[Yamada]

などの書籍を参考にまたは引用しました。

「あ～お」

RNA リボ核酸 (ribonucleic acid) の総称。(塩基) - (D-リボース) - (燐酸) からなるヌクレオチドが鎖状に重合したポリヌクレオチド。すべての細胞の核および細胞質でタンパク質と結合し、あるいは遊離の状態で存在する。リボソームを構成するリボソーム RNA(rRNA)、メッセンジャー RNA(mRNA)、転移 RNA(tRNA) など、いろいろな種類がある。

RNA ワールド 原始地球上で、複製と進化という生命の最も基本的な活動が RNA だけによってなされていたと想定される時代のこと。従来生命の起源についてタンパク質が先か核酸が先か、という問があり、両者が相互依存的であるため解答が見い出されなかったが、テトラヒメナの rRNA のイントロンが自己スプライシングを起こす事が発見され、さらに同じイントロンに C ポリメラーゼ活性を導入できる事が証明されてから、RNA が RNA だけで酵素活性を持ちうるという考えが一般的となり、生命の起源として RNA ワールドという考え方が定着した。

遺伝暗号、遺伝コード (genetic code) タンパク質のアミノ酸配列に対する遺伝情報を、核酸上にその塩基配列として記述するための暗号 (コード)。遺伝情報の担い手は核酸であり、核酸の塩基配列をもとにタンパク質が合成されている。

遺伝子 (gene) 遺伝形質を規定する因子。遺伝子は自己増殖し、細胞世代、個体世代を通じて親から子に受け継がれ、形質発現に対する遺伝情報を伝達する。各々の遺伝子は互いに独立の単位であるが、物理的に独立して存在しているのではなく、染色体上にそれぞれ固有の位置を占め、一般には線状に配列して連鎖群を形成している。ただし、まれに互いに重複して存在している場合がある。遺伝子は安定であるが、まれに突然変異や遺伝子組換えにより変化し、それが次世代に伝えられる。

遺伝子の本体は DNA (一部のウイルスでは RNA) であり、そのヌクレオチド配列の特異性によって、個々の遺伝子が規定される。すなわち遺伝子とは核酸分子上のある長さをもった特定の領域 (ドメイン) を指すことになる。原核生物の場合、例えば、あるタンパク質に対する遺伝子とはそのタンパク質の一次構造 (アミノ酸配列) に対応するヌクレオチド配列を指し、翻訳の際の開始点と終止点とにはさまれた部分をいう。真核生物の場合は、翻訳されないイントロン部分によって分断されており、そららの部分は mRNA を合成する際、スプライシングとよばれる過程で除去される。

タンパク質や RNA などの一次構造を規定している遺伝子を特に構造遺伝子 (structual gene) とよび、普通、遺伝子というときには構造遺伝子のことをいう。他方 DNA には制御領域とよばれる特定の配列が存在し、例えばプロモーターやオペレーターなどのように、形質発現などの制御、調節に重要な役割を果たすものがある。

遺伝子型 (genotype、ゲノタイプ) 生物の遺伝子の構成の事。「表現型」と対置される。

遺伝的アルゴリズム (Genetic Algorithm, GA) 生物集団の進化過程を説明する自然淘汰説を、記号列の集合に適用し、適応度の高い記号列が進化していく状況をシミュレーションする方法。J.Holland が提唱した。以下のようなアルゴリズムである。

- (1) ある長さの記号列の集合をランダムにつくる。
- (2) それぞれの記号列に対し適応度を計算し、適応度に応じて次世代に残す記号列を決める。
- (3) 次世代に残す記号列の中からいくつかを選び、その記号列のある部分を変化させる (突然変異)。
- (4) 次世代に残す記号列の中からいくつかペアを選び、お互いの記号列のある部分を交換させて新しい記号列を作る (交叉)。
- (5)(2) に戻る。

遺伝的プログラミング(Genetic Programing, GP) スタンフォード大学のジョン・コーザ(Coza)博士が提唱した、進化的計算手法の一つ。遺伝的アルゴリズムでは遺伝操作対象となる記号列をどのように決めるかはっきりとした方法論がない。遺伝的プログラミングでは、プログラム・コードそのものを遺伝操作の対象とする。LISPという言語はS表現という木構造でプログラムされるが、その木構造の枝の部分を突然変異や他の木の枝と交換などをする事（交叉）により、新たなプログラムを生成する事ができる。現在ではLISP言語の他にC言語などでにもこの技術が適用され、活発な研究が行われている。

イントロン(intron) 遺伝子のうち、機能をもつ最終RNAに含まれない配列。一般に真核生物の遺伝子には最終的にタンパク質またはRNAとして発現するヌクレオチド配列であるエキソンが、発現しない特定のヌクレオチド配列によって分断されている場合が多い。遺伝子の発現過程では、一つの遺伝子を構成するイントロンとエキソンは連続した一本のRNAに転写され、次の段階でスプライシングによってイントロン部分が除去され、エキソン部分が再結合される。

エキソン(exon) イントロンを参照。

L-System(Lindenmayer's system) 植物の成長過程を表現する枠組となる記号の並列書き換えシステムとしてA.Lindenmayer(1925～1989)が提案した。例えば紅藻の一種の発生の様子は8回の細胞分裂で次図のようになる。

これを記号的に表現すると、

$$S1 \rightarrow 1$$
$$S2 \rightarrow 42$$
$$S3 \rightarrow 443$$
$$S4 \rightarrow 4453$$
$$S5 \rightarrow 447653$$
$$S7 \rightarrow 448(1)7653$$
$$S8 \rightarrow 448(42)8(1)7653$$

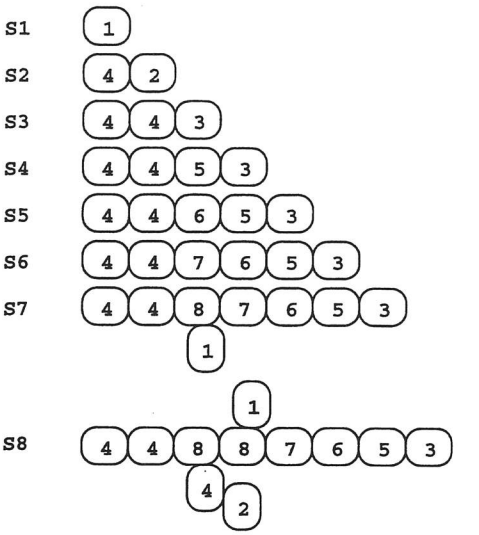


Figure A.1: リンデンマイヤーによる紅藻の一種の発生の模式図

のように書くことができる。ここで、8(1)は2つの記号(,)により、タイプ8からタイプ1の細胞が枝別れしてできたことを示す。

この発生過程において、ある型の細胞が別の型の細胞に変わるときの書き換え規則(Production Rule)Pを見い出すことができる。

$$\begin{aligned} P = \quad & \{1 \rightarrow 42 \\ & 2 \rightarrow 43 \\ & 3 \rightarrow 53 \\ & 4 \rightarrow 4 \\ & 6 \rightarrow 7 \\ & 7 \rightarrow 8(1) \\ & (\rightarrow (\\ &) \rightarrow)\} \end{aligned}$$

この規則Pおよび、細胞タイプを記号化したものの集合を

$$\sum = \{1, 2, 3, 4, 5, 6, 7, 8, (,)\}$$

とし、出発点の細胞を表す$\omega = 1$を合わせた組、

$$(\sum, P, \omega)$$

はこの紅藻の発生過程を数学的に記述することができ、このような組をL-systemという。

(核酸の)塩基(base) 核酸やヌクレオチドの塩基性の部分。プリン塩基とピリミジン塩基に大別される。前者にはアデニン、グアニン、後者にはシトシン、ウラシル、チミンがある。

オペランド(operand)、オペレーター(operator) コンピュータの命令語の構成要素で、命令の対象になる部分をオペランド、演算そのものをオペレーターという。例えば、X+Yという演算では、＋がオペレーター、xとyがオペランドとなる。

オペレーティング・システム(Operating System, OS) 計算機システム全体を基本的な部分で管理するソフトウエア。システム資源（メモリー、CPU、ディスク、入出力装置など）を各アプリケーションソフトが独自に管理するのは繁雑であるため、OSが代わりに基本的な管理を行う。各アプリケーションソフトはOSを通してシステム資源を利用する。現在パソコン用OSとしてはWindows95,98が主流なのはご存じの通り。他にMacOS,UNIX,最近台頭してきたLINUXなどが有名である。Tierraはこれらの本物のOSから見ればアプリケーションソフトの一つだが、Tierraの中で活動するデジタル生物たちからみると、仮想OSとしての役割を果たしている。

「か～こ」

開始コドン(initiation codon) mRNAがタンパク質に翻訳されるとき、タンパク質合成の開始点となるコドン。一般にはAUGが開始コドンであるが、GUGやUUGなど他のコドンが開始コドンとして使われる例もある。

核酸(nucleic acid) 塩基、糖、燐酸からなるヌクレオチドが重合した長い鎖状の分子。DNAとRNAに大別される。

カンブリア爆発(Cambrian explosion) 約5.6億年前から約5億年前と推定される古生代の最も古い時代をカンブリア紀(Cambrian period)という。三葉虫の全盛期だが、他のあらゆる動物の門も出そろった時代だと考えられている。これらの門は先カンブリア代の後期に起源したのち、この頃急速に分化し生態的空白を満たすように進化したと考えられ、これをカンブリア爆発とよんでいる。

機械学習(machine learnig) 人工知能の分野で、計算機に学習能力を持たせる事を目的とする研究。初期にはパターン認識の分野で発達した。次に学習内容を数値表現から記号に表現されたものへ移行した概念学習や言語獲得の研究が始まった。

その後、人工知能の応用として知識工学が発展し、ある領域に依存した専門知識をエキスパートシステムとして実現した。典型的なエキスパートシステムはプロダクションシステムとよばれる推論システムをIF THEN形式のルールベースとして持っている。ただし、エキスパートシステムでは専門家の知識を明示的かつ形式的に記述しなければならない。専門知識には“勘”のように言葉にしにくいものがあり、知識獲得のボトルネックとなる。

機械学習は例さえ与えられれば、ルールを学習し、知識獲得を自動化できる可能性がある。

さて機械学習では概念を学習する概念学習がその研究の大部分を占める。具体的な表現で概念を記述したものが、概念記述とよばれる。一般に機械学習には、帰納学習、演繹学習、発見的学習、類推学習、記憶に基づく学習などに分類され、多くの研究が行われてきた。

近年機械学習は、学習主体（エージェント）とその外界との相互作用の改善（適応）という視点から、適応エージェントやそれを集団化して進化的な方法（分類子システム、遺伝的アルゴリズムなど）を導入させる事などにより、いくつかの新たな研究分野を形成している。

機械語 コンピュータが直接理解し、実行可

能なプログラム言語。2進数により表現されるそのコンピュータ(CPU)独自の命令セットにより構成させる。機械語を人間にわかりやすい記号（ニーモニックコード）で書き換えたものをアセンブラ言語という。

寄生(parasitism)　共生の、普通はそれによって寄生者が利益を受ける片利共生の一形態。養分を搾取される側を寄主（宿主(host)）、搾取する側を寄生者(parasite)とよぶ。

逆ポーランド記法(Reverse Polish Notation)　スタックを使って計算することを前提とした記法。引数を先に書きその引数に対する演算子を後に書く記法。

例えば、(a＋b)ならab＋と書く。
また、(a＋b)×cならab＋c×となる。
c×(a＋b)ならcab＋×となる。

最後の例で、a=1,b=2,c=3のとき、スタックのイメージは以下のように操作される。

```
式          スタック
c          : 3
c a        : 3 1
c a b      : 3 1 2
c a b +    : 3 3
c a b + ×  : 9
```

最後にスタックトップに残っている値が、式の値となる。

ゲノム(genom(e))　染色体または遺伝子全体を呼称する語。

共生(symbiosis)　異種の生物が一緒に生活している現象。互いに行動的あるいは生理的に緊密な結び付きを定常に保っている場合を意味する。同じ生息場所に住んでいるだけではこの概念には入らない。

欠失(deletion)　突然変異の一つで、遺伝子（または染色体）の一部分が欠落すること。

交叉(crossingover)　相同染色体部分体間に生ずる部分交換の現象。結果として遺伝子の組換えが起こる。遺伝的アルゴリズムでは、2つの個体間で遺伝子に相当する記号の部分列を交換して新しい記号列を作る事をいう。

コドン(codon)　遺伝暗号の単位。核酸(mRNA)を構成している4種の塩基（アデニン、グアニン、シトシン、ウラシル）のうちの3個の配列が単位となって、それぞれのアミノ酸に対応している（トリプレット暗号）。64通りの組合せが可能だがそのうち61通りがアミノ酸のコドン、残りの3つが終止コドンとなっている。

「さ～そ」

サブルーチン(subroutine)　プログラムの補助的なひとまとまり。補助的なルーチンであり、メインとなるルーチンから何度も呼び出されることがある。

細胞系譜(cell lineage)　受精卵から成体ができるまでの発生過程において、細胞がどのように分裂し、どのような組織細胞に分化するかを記述したもの。近年、線虫の一種(C.elegans)では受精卵から成体にいたるまでの完全な細胞系譜が解明された。

細胞膜(cell membrane)　細胞膜は原型質を包む脂質二重層であり、半透膜性をもっている。

C言語　1972年、米国AT＆Tベル研究所でデニス・リッチーが開発した関数型の手続き言語。DECのミニコンPDP-11上で動くOS、UNIXを記述するために開発された。実践的なプログラムを処理系に依存しないで書くことに優れている。処理系によってはメモリやレジスタを直接アクセスできるので、高機能なアセンブリ言語ともいえる。現在、パソコンからスーパーコンピュータまで幅広い分野で使用されている。Tierra自体のプログラムソースはC言語で記述されている。

CPU(Central Processing Unit, 中央演算処理装置) コンピューターのすべての装置の動作を統合制御し、さらには算術論理演算を実行するもの。1個のLSIで機能するCPUをマイクロプロセッサという。算術論理演算を実行する部分をとくにALU(Arithmetic and Logic Unit)と呼ぶ。Tierraにおいては各デジタル生物たち各々が一つまたは複数個の仮想CPUを持つものとし、遺伝コード（プログラム）を実行する。

終止コドン(termination codon) mRNAがタンパク質に翻訳されるときタンパク質合成の終止を指示するコドン。一般にはどのアミノ酸にも対応しないコドンすなわちナンセンスコドンUAA,UAG,UGAがこれに相当し、これらの一つまたは二つの組合せでタンパク質合成の終止点が規定される。タンパク質をコードしているmRNAの上に終止コドンが表れると、リボソーム上でのペプチド鎖合成はその位置で停止する。さらに、終止コドンに対応したポリペプチド鎖終止因子が作用してペプチド鎖がリボソームから遊離し、その合成がストップする。

進化生物学(Evolutionary Biology) 進化生物学は、進化の機構とプロセスを分析し、個々の生物集団と生命の進化の歴史を推定する事を目的としている。ある生物群に着目し、その系統や生物地理、進化のパターンを研究する。しかし最近は進化の「総合説」に立脚し「集団遺伝学」や「分子生物学」を駆使して、生物一般の進化プロセスの研究に重点が移っている。

進化論(evolution theory) 生物の種が世代交代を重ねるうちに別の種へと変化していく現象を進化とよぶ。そして、その現象の原因を究明するものが進化論である。主な進化論には以下のようなものがある。

- ラマルキズム
  フランスのJ.ド.ラマルク(1744～1829)が唱えたもので二つの原理に基づいている。一つは「用不用説」と呼ばれ、動植物の器官のうち使用頻度の高い有用な器官は進化して強大となり、逆に利用されなくなった物は退化するという説。そして第2の原理が「獲得形質の遺伝」である。これは環境が種に及ぼした「用不用」原理に基づく変化が、子孫に伝わるというものである。ただし、その当時は遺伝子の概念自体がなく、これらの原理はラマルクの類推である。そのかわりラマルクが進化の根本原因と考えたのは、生物は環境の変化にかかわらず、本来複雑高度に進化する内的要因を持っていたとする、「漸進的発達」説である。
- ダーウィニズム
  C.ダーウィン(1809～1882)が「種の起源」で著した今日の進化論の基本となる説。種個体群の中に優劣の個体差ができ、そのうちで環境に適応できた個体が生き残るという「自然淘汰」説を唱えた。ただし、彼はまだ突然変異には気付いておらず、進化とはあくまで種の中のわずかな個体差が長い時間をかけてふるいわけされる過程と考えていた。
- 総合説
  獲得形質の遺伝を否定し、遺伝性の飛躍的な変異、すなわち突然変異と外部環境を厳密に形質の優劣判定のふるいとしてのみ規定する自然淘汰を組み合わせて作られたのが新ダーウィニズムまたは総合説という。この説は19世紀末から20世紀初頭に急速に進歩した「集団遺伝学」によって進化論の主流になった。変異遺伝子の個体群内での定着率の数式化など遺伝子の拡散・固定のプロセスを定量的に取り扱う手法が確立した。また、遺伝子の機構解明を目指す「分子生物学」の発展もこの説を側面から支えた。
- その他、自然淘汰説に反対する非ダーウィン進化や、遺伝子重複説、分子進化中立説、住み分け理論など多くの重要な理論があるがここでは省略する。

人工生命 (artificial life)　生命現象の主として情報的側面を非有機的あるいは有機的人工物として構成的に捉えたもの。人工生命の典型的な舞台はコンピュータシステムである。その中で記号の列や、種々の図形などで表現される人工生命体が、交配・増殖・運動・学習・適応・進化といった生命諸活動を行う。このような生命のような人工物を構築し、そのありさまをシミュレーションなどによってあきらかにする事によって、現存する生命の諸側面の理解を深めようとする“弱い人工生命”という立場がある。それに対して、およそ生命とよびうるものを構築しその挙動を解明する事を人工生命研究の目的とする考え方“強い人工生命”の立場がある。人工生命の言葉が広く使われるようになったのは1987年の人工生命国際会議以降である。この会議を組織したC.Langtonらは、人工生命を特徴づけるキーコンセプトとして創発 (emergence) を挙げている。これは下位レベルの要素間の局所的相互作用が一つ上のレベルの新たな大域的状態を生み出す事を指す。

スプライシング (splicing)　真核生物のDNAには遺伝情報を有している部分（エキソン）の他に、意味のない、イントロンと呼ばれる配列がある。これを除去し意味のあるエキソンだけからなるmRNAを作るため核内で進行する過程。

セグメント・メモリ (segment memory)　8086CPUは、20ビットからなるアドレスバスを持ち、1Mバイトまでのメモリを扱う事ができる。そしてメモリ空間の中から一つのメモリを指定する方法として「セグメント方式」いうものを用いる。1Mバイトのメモリ空間をセグメントという単位に細かく区切って管理することにより、メモリの無駄な使い方を省いたり、高速化が実現される。

アドレスを物理アドレスで指定するかわりに、セグメントアドレスとオフセットアドレスの2つの数字で指定する。セグメントアドレスは物理アドレスの中で、そのセグメントの基準位置（セグメントベース）を決めるためのもので、オフセットアドレスはセグメントベースを基準にしたアドレスである。

この方式は80X86シリーズの上位のCPUでも受け継がれている。

染色体 (chromosome)　動植物細胞の有糸分裂の際に観察される塩基性色素で濃く染まる棒状の構造体。現在では細胞小器官にある線状配列の遺伝子連鎖群をも広い意味で染色体と呼ぶ。染色体の数や形は生物の種によって特異的であり、細胞分裂時の核分裂の際に極めて複雑な行動をする。真核生物の染色体は、DNAとヒストンなどの塩基性タンパク質を主体とし、これにRNAや酸性タンパク質などが加わってできている。

挿入（insertion）　遺伝子の組み込み。ファージゲノムやプラスミドなどが他の染色体の中に組み込まれること。遺伝的アルゴリズムにおいては遺伝子を構成する記号列に別の記号列を挿入することをいう。

「た～と」

重複（遺伝子重複,Gene Duplication）　ゲノム内に同じ遺伝子が2個またはそれ以上存在すること。遺伝子重複は、遺伝子量の増加や、遺伝子の位置効果による形質発現への特異的影響などによって生物の適応値を高める場合がある。高等生物のrRNA遺伝子などでは極めて多数の同一遺伝子がならんでいる場合がある。

TCP/IP(Transmission Control Protocol / Internet Protocol)　TCP/IPはインターネットにおいてもっとも多く使われている通信手順である。非接続型パケット配信 (connectionless packet delivery) と高信頼ストリーム転送 (reliable strem transport という2つの基本的なプログラム手法を提供する。C言語ではsocketとよぶシステムコールによって実装できる。

突然変異　ゲノムを構成する染色体や核酸分子の一部に生じる永続的な変化の事。DNA分

子に変化を生じた部位を突然変異部位とよぶ。あるヌクレオチドが別のヌクレオチドに置き変わったものを点突然変異とよび、それによりコドンが変化する。欠失や挿入などの起こる単位は数塩基から長大なDNA領域にわたる場合もある。染色体レベルでの逆位や転座という突然変異では、遺伝情報の位置関係がかわり、大きな構造変化をもたらすので生存に有害な結果をもたらすといわれる。突然変異は体細胞にも生殖細胞にも起こるが、生物進化の上で重要なのは生殖細胞に生じた後代に伝達されるものである。したがって一般には体細胞突然変異が後代に伝達される事はなく獲得形質は遺伝しない。

「な～の」

Nice値 UNIXシステムでCPU使用権の優先度を決めるのに使用される値。nice値は小さいほどCPU使用権が高く、長い間連続してCPUを使用することができる。この値は0～39の間をとる。特に何も使用しない場合（デフォルト）では20である。

Tierraシステムではnice値を最低の39に設定される。よって、コンピュータシステムの負荷によってTierra内のデジタル生物たちのCPU実行速度が変化しやすくなる。

二倍体(diploid) 生活環の中の相同または異種の染色体を2組もつ時期の細胞または個体。

ヌクレオチド(nucleotide) 塩基-糖-燐酸からなる物質。塩基には4種類あるのでヌクレオチドにも4種類ある。これらがたくさん鎖状につながってDNAやRNAを構成する。

「は～ほ」

VRML(Virtual Reality Modeling Language) WWWで3次元モデルを表示するための言語。3次元空間内にHTMLと同様の“リンク”という概念を導入し，仮想現実の構築を目指した言語である。

フォン・ノイマン型コンピュータ(Von Neumann type computer) 現在使われているコンピュータのほとんどはこの型のコンピュータである。プログラムを内蔵した逐次制御型コンピュータである。

複雑系(complex systems) たくさんの要素が集まり、それが一個で存在しているときとは異なった新しい性質や能力を持つようになったシステム。

流体の運動は不規則だが有る程度予測可能である。このような、ある法則で時間発展する系をカオスという。

脳細胞は一個では高度な思考力を持たないが、脳というシステムになる事によって、思考力を持つ。要素個々の性質からは、ただちに推測できないようなシステムとしての新しい性質・能力が生じる事を創発という。

複雑系の研究は、システムのすべての現象はそれを構成する要素の性質で説明できるとする要素還元主義的な研究と対極の関係にある。

プロセス(process) 計算機システムにおける、処理過程の単位であり、実行中のプログラムをいう。Tierraにおいては個々のデジタル生物たちは、それらの遺伝コード（プログラム）を実行する仮想プロセスとして扱われる。

分化(differentiation)（細胞分化） いくつかの意味があるがここでは、個体発生しつつある生物の中で形態的・機能的に特殊化が進行し、特異性が確立される過程のこと。多細胞生物において1個の細胞の分裂によって由来した娘細胞集団の中で形態的・機能的に質的な差をもった二つ以上のタイプの細胞が生じてくる現象。ゲノム中の特定の遺伝子群が発現した状態。

分類子システム(Classifier System) 遺伝子としてコーディングされたルールを遺伝的アルゴリズムを用いて、ある評価関数の値を高くするようなプロダクションシステムを自動生成

するシステム。

環境からの情報をメッセージとして受取り、それに関係のある分類子があると、環境への行為が実行される。分類子システムの特長は分類子の適用により、問題を解決することができればその分類子は強化されることで学習が進み、また遺伝的アルゴリズムによりルールを発見することである。

「ま～も」

マルチタスク(multitask)、マルチスレッド multithread　一度に多数のタスクを処理できるシステムのこと。タスクとはとは計算機システムから見たプロセスの管理の単位である。人間（ユーザー）から見ると複数のタスクの集まりを一つのジョブとして計算機に仕事させることになる。また、マルチタスクにおいて同時に処理されるのひとつひとつのタスクをスレッドと呼ぶ。

命令セット(instruction set)　コンピュータに計算その他の処理を知らせる指図を命令という。一つの命令は、処理の種類と処理の対象との組合せである。処理の種類はコンピュータによって異なっている。そのコンピュータで使用できる命令をまとめて命令セットと呼ぶ。

メッセンジャーRNA(mRNA)　遺伝子の情報がタンパク質として発現される過程で、情報の担体として合成されるRNA。ゲノム上の遺伝情報は一定の単位でRNAに転写される。その際、DNA依存性RNAポリメラーゼはDNA上のプロモーター部位を認識してこれと結合し、特定の位置からそのDNA鎖の一方を転写して、これと相補的なRNA鎖を5'側から3'の方向に合成する。RNAポリメラーゼはターミネーター領域にくると転写終結しDNAから離れる。こうして遺伝子DNAの一方の鎖のヌクレオチド配列と相補的な、一連のRNAが合成される。原核生物ではほとんどの場合、複数個の遺伝子が一つながりのmRNAとして転写される。真核生物の場合は、種々のプロセッシング過程（スプライシングなど）を経た後、核から細胞質へ移動し、はじめてmRNAとしての機能をはたす。真核生物のmRNAは一般に各遺伝子ごとに形成される場合が多い。

MIMD(Multiple Instruction-stream Mutiple Data-stream)型並列コンピュータ並列コンピュータで、それぞれの各プロセッシング・ユニットが、それぞれに命令を実行処理するタイプ。これに対してすべてのプロセッシング・ユニットが単一命令の処理を同時に行うタイプをSIMD(Single Instructin Multiple Data-stream)という。

「ら～ろ」

RAM (Random Access Memory)　ランダムにアクセスできるメモリ。読み込み書き込み可能なメモリ。通常電源を切ると内容は消えてしまう。パソコンのメインメモリとして使用されている。Tierraでは、デジタル生物のプログラムコードを置くためのメモリを用意している。これをスープメモリと呼んでいる。これは本物のメモリの一部をTierraが確保したものである。本物のメモリ(RAM)が大容量ならそれだけ、スープメモリも大きくでき、より大規模ばシミュレーションが可能になる。

Lisp言語　1960年、MITのJ.マッカシーが開発した。研究用に広く用いられた。S表現という木構造でプログラムを作成していく。知能を表現するのにリスト構造が適していたためとくに人工知能の研究で重要な役割を果たした。

レジスタ(register)　置数器といわれるもので、数を保持し、いつでもその内容を利用できる。使用目的により、汎用レジスタ、インデックスレジスタ、フラグレジスタなどがある。

Lotka-Volterraサイクル（ロトカ-ヴォルテラ・サイクル）　披食者-補食者などの間で、

それぞれの個体数がある周期ずれながら増減を繰り返すサイクル。Lotka-Volterra 方程式でモデル化される。実際の生物個体群間の相互作用にそのままでは適合しないが、種間競争や補食作用による個体群の本質を捉えている基本的モデルである。

「わ」

ワーム (Worms)　独立したコンピュータプログラムでコンピュータウィルスと同様に自己増殖する。コンピュータウィルスとの違いは、実行可能なプログラムなしで独力で自己増殖できる点である。1988 年 11 月に出現した「インターネット・ワーム」は多くの大学や研究機関に進入し、大混乱を起こした。

参考文献

[ALife] Edited By Christopher G.Langton. 1989. Artificial Life. Addison-Wesley Publishing Company.

[Charrel95] Charrel, Agness. 1995. Tierra network version. ATR Technical Report TR-H-145.

[Doi63] 土居洋文. 昭和63年. 生物のかたちづくり. 発行 株式会社サイエンス社

[Futuyma] 進化生物学（原書第2版）、1991. 岸由二他訳、発行 蒼樹書房

[Goldberg89] David E.Goldberg, 1989. Genetic Algorithms in Search, Optimization, and Machine Learning, pp217-276, ISBN 0-201-15767-5, Addison-Wesley

[Holland78] Holland,J.H.,and Reitman,J.S. 1978. Cognitive systems based on adaptive algorithms. In D.A. Waterman and F.Hayes-Roth(Eds.), Pattern directed interference systems(pp.319-329). New York:Academic Press

[Iwanami] 岩波生物学事典 第4版. 1996. 八杉龍一他編集、発行 岩波書店

[Kimezawa94] Kimezawa Tsukasa. 1994. CM-5 利用の手引
http://www.hip.atr.co.jp/~kim/CM5/Guide_header.html

[Kimezawa96] Kimezawa Tsukasa. 1996. Beagle,(New Tierra Front end Tool).
http://www.hip.atr.co.jp/~kim/beagle/beagle.html

[Kimezawa97] 木目沢司、和田健之介、トーマスレイ、下原勝憲. 1997. デジタル生態系における多細胞生物の進化、情報処理学会第55回（平成9年後期）全国大会講演論文集（2）:439-440

[Koza92] Koza, John R. 1992. Genetic programming, on the programming of computers by means of natural selection. Cambridge,MA: MIT Press.

[Lindenmayer89] Aristid Lindenmayer and Przemyslaw Prusinkiewicz, "Developmental Models of Multicelluar Organisms: A Computer Graphics Perspective," ARTIFICIAL LIFE, edited by C.Langton, pp221-249, Addison-Wesley, 1989

[Stuart] Stuart Russel,Peter Norvig 著、古川康一監訳. 1997 エージェントアプローチ人工知能. 発行 共立出版

[Ray91] Ray, T. S. 1991. An approach to the synthesis of life. In : Langton, C., C. Taylor, J. D. Farmer, & S. Rasmussen [eds], Artificial Life II, Santa Fe Institute Studies in the Sciences of Complexity, vol. XI, 371-408. Redwood City, CA: Addison-Wesley.

[Ray92] Ray, T. S. 1992. Evolution, ecology and optimization of digital organisms Santa Fe Institute working paper 92-08-042.

[Ray93] Ray, T.S. 1993. How I created life in a virtual universe.

[Ray94] Ray, T. S. 1994. Evolution,complexity,entropy,and artificial reality. Physica D 75:239-263.

[Ray95a] Ray, T. S. 1995. A proposal to create a network-wide biodiversity reserve for digital organisms. ATR Technical Report TR-H-133.

[Ray95b] Ray, T. S. 1995. An evolutionary approach to synthetic biology: Zen and the art of creating life. In: Langton, C. G. [ed.], Artificial Life, an overview. The MIT Press, 1995. Reprinted from: Artificial Life 1(1/2): 195-226.

[Ray98] Thomas S.Ray & Joseph Hart, 1998. Evolution of Differentiated Multi-threaded Digital Organisms, Artificial Life VI proceedings.

[Ray98b] Thomas S.Ray, 1998. Tierra.doc 17-3-98 Documentation for the Tierra Simulator , http://www.hip.atr.co.jp/~ray/pubs/doc/doc.html

[RayNetReport] Continuing Report on the Network Tierra Experiment http://www.hip.atr.co.jp/~ray/tierra/netreport/netreport.html

[Roger] Roger Lewin "Life in a Computer" In: Complexity LIFE AT THE EDGE OF CHAOS, Chapter 5, pp.84-105.Macmillan Publishing Company
（邦訳）コンプレクシティへの招待 - 複雑性の科学 1993, ロジャー・リューイン（著）、福田素子（訳）、徳間書店

[坂無 94] 坂無, 鈴木 and 嘉数, 1994. エコ・プログラミングへのアプローチ -TSP- への適用 -, 日本機械学会第 71 期通常総会講演会論文集 (I), pp.124-125.

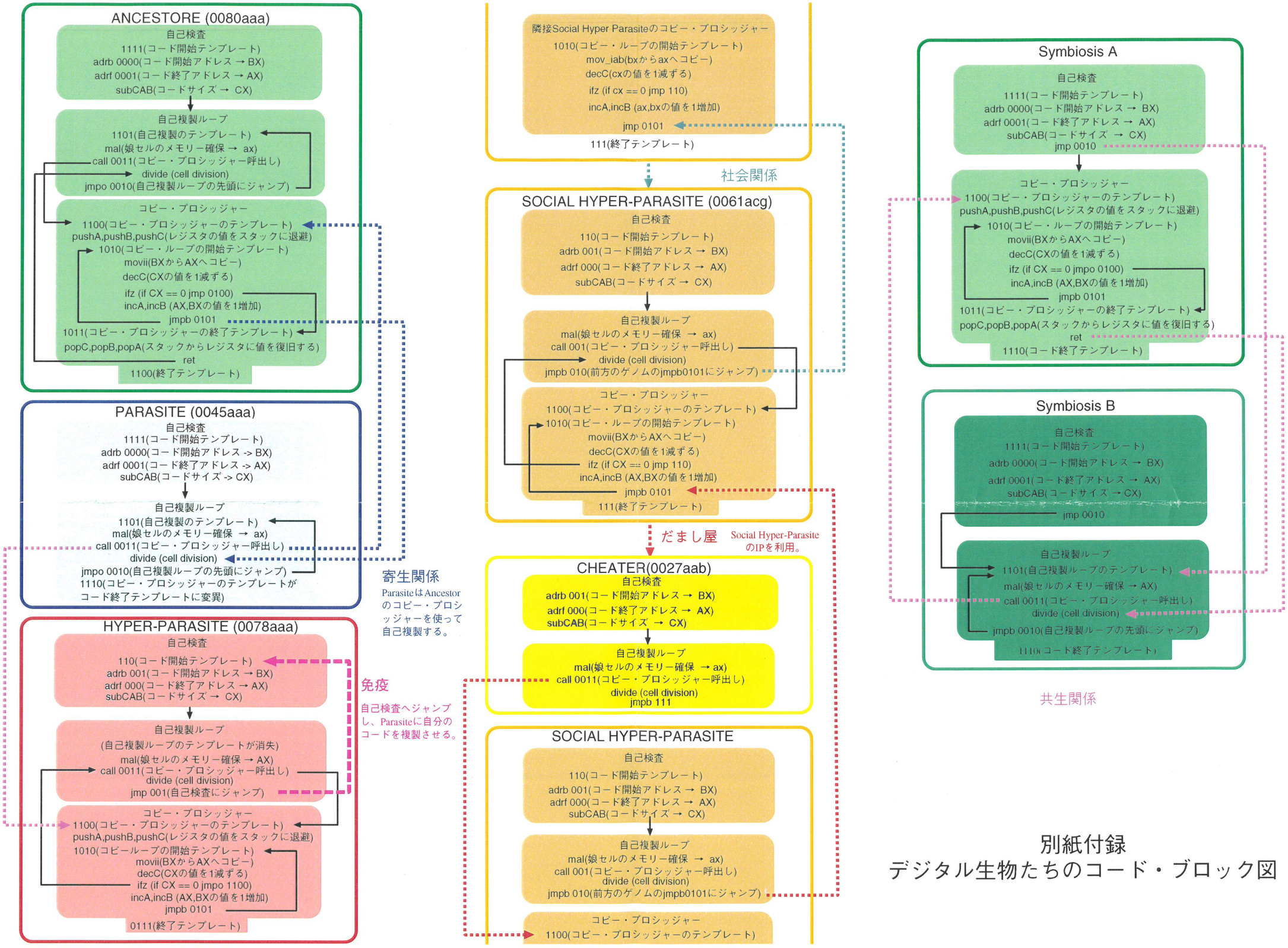
[Sims91] Sims, K. 1991. "Artificial Evolution for Computer Graphics," Computer Graphics(Siggraph '91 proceedings), Vol.25,No4,July 1991,pp.319-328.

[Thearling94] Thearling,Kurt and Ray, T. S. 1994. Evolving multi-celluar artificial life. Brooks,Rodney A.,and Pattie Maes[eds.] Artificial Life IV conference proceedings, Pp.283-288. The MIT Press, Cambridge.

[TierraWorkShop95] Tom Ray, Jeremy Bruestle, Roger Gouin, Joseph Hart, Matt Jones, Kurt Thearling, Jan Hauser, Charles Ofria, and Titus Brown. 1995 Tierra workshop report. Unpublished (Please see Tom Ray's Home page).

[TierraWorkShop96] Tom Ray, Hayward R Alker, Manor Askenazi, Jennifer Cobb, Tarek Elaydi, Linda Feferman, Simon Fraser, Gilly Furse, Joseph Hart, Jan Hauser, Matt Jones, Tsukasa Kimezawa, Will Rose, Walter Tackett, Kurt Thearling. 1996 Tierra workshop report Unpublished (Please see Tom Ray's Home page).

[Yamada] 認知科学モノグラフ 8 適応エージェント. 1997 山田誠二著. 発行 共立出版



別紙付録
デジタル生物たちのコード・ブロック図