記号処理1-3
(1977. 7. 21)

ゲーム・プログラミングの手法

竹内郁雄(電電公社武蔵野電気通信研究所)

はじめに

ゲームを競技するプログラム(以下ゲーム・プログラムと呼ぶ)を作成すること(ゲーム・プログラミング)は非常に興味深い。ある意味ではゲームを競技することよりも興味深いと言える。昨今のマイクロコンピュータの流行でこの楽しみを実感として味わえる機会が多くの人々に与えられたのは御同慶の至りである。(もっともマイコンでは力不足の感は否めない) ゲーム・プログラミングが何故興味深いのか以下理由を列挙してみよう。

1. 人工知能研究の中では最も早くから成功をおさめたものである。なにしろ計算機がこの世に登場する前から多くの著名な人々によってゲーム・プログラミングは議論されていた。
2. 人工知能研究の中では珍らしく人間と機械の出発点が同じである。つまりゲームは人間にとっても知的活動を強いるものであり,他の分野,自然言語理解やパターン認識のように人間にとって自然に身についているものを計算機で苦労してやるのとは違い,人間と計算機はゲームのルールというある意味では抽象的な(人生経験に左右されないような)共通の出発点に立つ。この意味では定理証明も同じであるが,これにしても定理の背後の意味の直観的理解という日常的雑多が紛れ込んでいる。
3. ゲーム・プログラムは"人格"を持つ。ゲームがもともと人間の闘争本能に根ざす以上,勝ち負けという人間的関係を通じてゲーム・プログラムと対戦する人間が知らず知らずのうちに相手(計算機)を人格化して扱うのは当然のことかも知れない。itがheに代わり,「計算している」が「考えている」になり,「おぬし出来る!」が連発されるようになるのは誰しも経験することである。
4. プログラムの性能評価が勝負によって決定される。つまり評価基準が極めてはっきりしている。ゲーム・プログラミングの泥沼的魅力は主としてここにある。
5. プログラムの重要な部分は非数値計算である。つまり記号処理のよいアプリケーションである。人間がゲームを競技する場合,恐らく100以上の大きな数は無限大と同じである。(おお忌しき大数値!)
6. ゲームの理論はあるにはあるが,いくら計算機が速くなっても必勝の手を打つプログラムが存在し得ない位に,ゲームの木の深索は膨大で時間がかかる。よってtrivialでないゲーム・プログラムに残された道はnear-optimalを求めることのみである。"near-optimal"——これほどつかみどころなき興味尽きぬ対象はない。プロ棋士は日夜このために苦労しているのだ。
7. プログラム技法上,面白い問題を多く含んでいる。

これらのうちいくつかについては第3節でさらに詳しく述べられるであろうが逆に上に述べた点を満足し得ないようなゲーム・プログラムは議論の対象外である。たとえば必勝法のわかったゲームのプログラムを組むこと,これはプログラミングの初等練習問題になりこそはすれお世辞にも興味ある問題とは言えまい。

またある種の数あてゲーム（マイコンでは主流の一つである）のように機械の得意な分野で全数探索をやってoptimal（必勝とは限らない）な戦略をとれるようなものも議論の対象外である．逆に19×19の正調囲碁を一手数秒以内で打つようなプログラムを現在の"技術"水準（計算機の能力というより，ゲーム・プログラミング技法上の水準）で考えることもどだい無理があると思われる．（チェスのゲームの木が10^{120}なら，囲碁は10^{760}と言われている．）要は計算機（言語）の性能とゲームの規模・性質の間に微妙なバランス関係がなければならないのであって，そのようなゲームを見付けること自体が最初の問題である．（ゲームの規模・性質の見積りは多少の経験が必要であるがそれほど難かしいことではない．ただ世の中には最初から無謀なことを試みる人もいないではない．）無論，良いゲーム・プログラミングの手法が開発されれば，計算機側の性能を一定にしたままでもプログラム可能なゲームの規模は増やすことが出来よう．この場合その増加率は，constant factorというよりorderで効く可能性がある．

第1節，第2節では筆者が最近作成したゲーム・プログラムHexとCalculationについてやや詳しく述べる．これらは完全とは言えないまでも筆者の主張を具現した実例であって，いくつかの点で読者諸兄の参考になるかも知れない．（プログラム言語は筆者及び奥乃博氏によって開発されたリスト処理言語LIPQである）第3節では筆者のゲーム・プログラミングの考え方についてある程度抽象的に述べる．

1. Hexプログラム

Hexは今世紀に入って発明されたゲームの中では最も面白いものの一つであろう．ルールは至極簡単である．図1のような盤面の向かいあった辺を各々黒陣，白陣とする．黒石を持つ競技者と白石を持つ競技者が交互に自分の石を置いていき，先に自陣の対辺を自分の石で継いだ方が勝ちである．なお石は六角のマスの中に打つ．

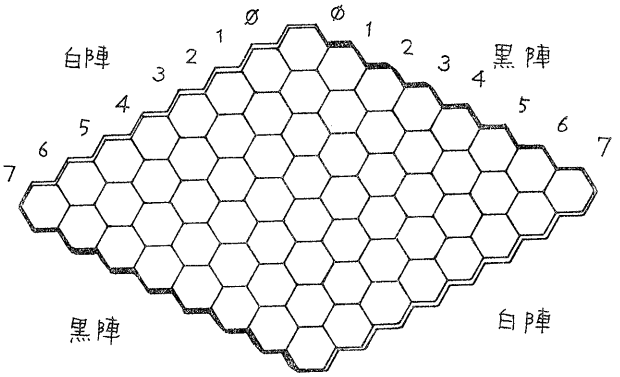


図1. Hexの盤面
（座標は左上の数字を先に読む約束）

図1の盤面は一辺が8としてあるが，人間同志が実際に競技する場合は一辺を11ないし12にしたほうが面白い．ここでは計算機の能力などを考えて，ゲームの面白さを失なわない最小の盤にしてある．Hexは先手必勝であることがわかっているが，その証明は非構成的なので実際の役には立たない．ただ先手がかなり有利に勝負を進められるのは事実で，そのため先手にハンディをつけることが多く行なわれる．（こうすれば先手必勝の証明はすでに成立しない）ここでは，先手（黒）の第1手を短かい対角線上に打つことを禁じている．

筆者のHexプログラムは，HexがGPCC（計算機によるゲーム・パズル競技会）の課題となったときに作成したもので，LIPQで1000行前後（アセンブラ語で書かれた今――モデム・インタフェイス，乱数ルーティン等がこれに加わる）である．（ディスプレイを介して競技する機能は野村浩郷氏に負う）

Hexの盤面は8×8の場合，8×8の2ビット整数配列があれば完全に表現出来る．しかしこれは実際の盤面との対応が簡単だという以外に何のメリットもない．重要なことは，盤面の表現が戦術に役立つ情報を抽出し易い形になっていることである．上に述べた配列表現はHexのルールすら知らない人の網膜レベルの認識を表わすに過ぎず，名人が一瞬にして脳裏に浮かべる盤面認識には程遠い．Hexの盤面を刻々変化していくグラフ構造として表現すれば，名人の域には達しないまでも，少しましである．連結の情報が直接埋め込まれるから，Hexのルールによって規定される盤面のsemanticsを内包させることが出来る．たとえば，同色の石の連鎖（ダンゴ）が出来たらそれを一点に縮退させ，白石と黒石が隣り合ったらその間の枝を取り去ることをやれば，Hexのルールに合致した無駄のない表現が出来る．これによってある一群の石が厚いというのは，対応する節点から自由空間に伸びる枝の数が多いというふうに言い換えることが可能である．

筆者のHexプログラムでは盤面の初期状態として，元の盤面と双対の関係にある図2のグラフを考える．1個石が打たれる度に，枝の消去，節点の縮退等によりこのグラフは変化していく．Hexのルールをこのグラフの言葉で置き換えれば「ゲームの目標は自陣の2つの特別な節点を最終的に同一点に縮退させること」になる．

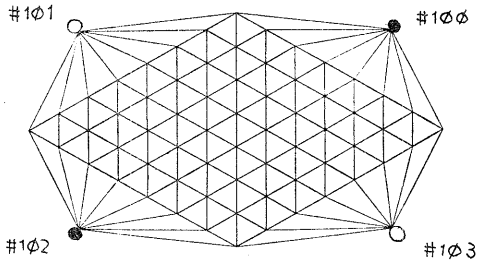


図2．盤の初期状態を表すグラフ

このグラフは68要素の一次元配列BOARDで表現される．座標ΦΦは第0要素，Φ1は第1要素，…，77は第#77要素，黒陣を#1ΦΦと#1Φ2，白陣を#1Φ1と#1Φ3の要素に対応させる．（#はLISPにおける8進数マークである．）各要素は1個の四元セルで，carがマス目の状態（NILは空），cdrはその点と同一視されている節点の集合，cdrが周辺との連結情報を表わす．図3は初期状態における第#62要素の内容である．これによれば，座標62は63，73，72，…，52と直接隣り合っており，#1Φ2とは73と72を支持節点として繋がっていることがわかる．なお点aとbが点cとdを支持節点として繋がっているとは一般に図4のような状態を指す．すなわちaとbに自分の石がある場合，敵がcとdに続けて打たない限りその連結を防げることが出来ないというわけである．グラフが複雑に変化すれば，これ以外の支持節点のパターンも出て来るし，支持節点の数を3以上になり得る．このような自明の2股を最初から盤面の表現の中にとり込んでおくことは処理速度の向上のみならず戦術上からも大変重要なことである．当初はこれだけで十分であろうという予想としてプログラムを組み始めたのであるが，これがまったく話にならない位弱く，次に盤端における特殊な連結情報を導入することとなった．これは謂ゆる台形定理と呼ばれるもので，図5のような状況では石aは必ず盤端（つまり自陣）に連結出来る．これは

```
[NIL (#62)
 NIL .((#63)
       (#73)
       (#72)
       (#61)
       (#51)
       (#52)
       (#102 #73 #72)
       (#53 #63 #52)
       (#74 #73 #63)
       (#71 #61 #72)
       (#50 #51 #61)
       (#41 #52 #51) )]
```
図3．BOARDの要素の例

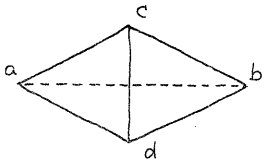


図4．第一階の2股

同じく一次元配列TRAPEZOIDで表現される．図6に初期状態におけるその第#52要素の内容を示しておく．この場合，"支持節点"の個数は8個である．この情報をBOARDに組み入れなかったのは理由がある．すなわち，台形定理は先の自明の2肢を使っておりその意味で第2階の2肢である．これを第1階の2肢と同一レベルで処理することは不可能である．いずれにせよ支持節点つきの連結情報は，支持節点のどれかが空でなくなれば消去される．(台形定理の場合，自分の色の石であれば消すには及ばない)

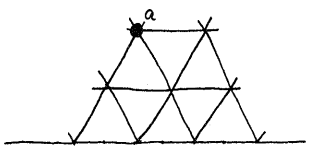


図5. 台形定理

```
(NIL (#101 #60 #50 #51 #40 #30 #61 #41 #62)
     (#101 #20 #30 #41 #40 #50 #31 #51 #42)
     (#102 #75 #74 #63 #73 #72 #64 #62 #53)
     (#102 #71 #72 #62 #73 #74 #61 #63 #51) )
```
図6.

以上の記述から推察される通り1個の石を盤面に置くためのプログラム量はかなり大きい．単純な配列表現において当該の要素に白ないしは黒を意味する情報を代入するだけと比べればまさに雲泥の差である．(LIPQで100行以上)ここで注意に値するのは，同一視された節点に対応するBOARDの要素は同じ四元セルを指すように変更されることである．(LISPでいうEQの意味で等しくなる)これによって連結成分が変化する毎に，代表点を選びなおしたりする必要がなく，任意の経路をたどってアクセスされた節点に対応するBOARD要素を変更すれば，その同一視節点(連結成分)に対応する要素の変更は皆同時に終了することが保証される．まさに配列データとリストデータの見事な協力関係であって，固定サイズの配列にもかかわらずこのグラフの動的な状態が自然に表現出来るのである．図8に図7の盤面におけるBOARD第#34要素(図7中◉に対応)の内容を示しておく．

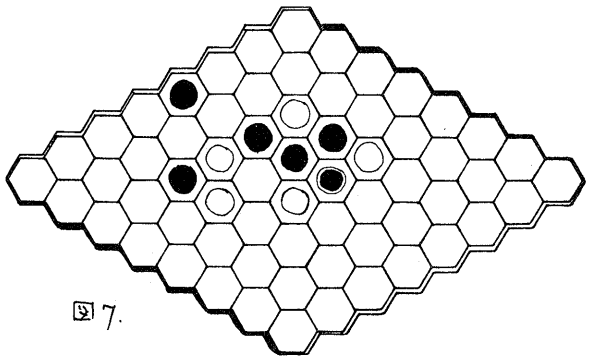


図7.

```
[BLACK (#23 #33 #34 #32)
 NIL . ((#12)
        (#13)
        (#20 #21 #31)
        (#21)
        (#31)
        (#35)
        (#43)
        (#45)
        (#46 #45 #35) )]
```
図8.

もちろん，BOARDとTRAPEZOIDだけでは，第2階までの連結情報を含んでいるとは言え，盤面を真に戦術的な眼で認識したことにはならない．たとえば自分の石が自陣に対してどの程度の連結性を持っているのか，あるいはどの一群の石が最も役に立っているか，火急の空地はどこかなどの情報があたかもあぶり出しのように自然に浮かびあがってくるようなより高度の盤面表現が必要である．すなわち盤面の上に飛び散る火花をを表現しなければならない．そのための一方法として，敵味方の総ての連結プラン(あと何手か打てば自陣が繋がるというパス)を列挙することを考えてみる．ただし連結プランは最少手で達成出来るものに限る．(プログラミング上この条件をはぶくと非常な困難が生ずる．)これは通常の最短経路を求めるアルゴリズムで大体事足りる．もちろん，支持節点つきの連結

があるため多少複雑なプログラムにはなる．たとえば一つの連結プランの中で同じ節点が2回以上支持節点に登場することは許されない．なぜなら敵はその点に石を打つことによってその連結プランを完全に破壊することが出来るからである．

図7の盤面においては黒の最少手数連結プランは10個あり，その手数は3である．図9にそのうちの3つを示す．#1ΦΦΦがかぶっているのはすでにそこに自分の石があることを意味し，括弧でくくってあるのは支持節点を示す．図9の一番上の連結プランは11，21，51の3箇所に打てば連結が完成すると読める．このような形で敵味方のプランを総て列挙し，それを盤面のもう一つ進んだ認識と考えれば，そこから肝要の点が浮かびあがってくるのが了解されるであろう．単純に考えて，敵味方のプランがもっとも多く交差するところが重要そうであると言えよう．（しかしこれは単純に過ぎる．）

```
(#1100 (#1 #0) #11 #21 (#31 #20) #1030 (#40 #41)
 #51   (#70 #71 #61 #72 #73 #60 #62 #50)
 #1102 )
(#1100 #0 (#11 #10) #21 (#31 #20) #1030 (#40 #41)
 #51       (#70 #71 #61 #72 #73 #60 #62 #50)
 #1102 )
(#1100 (#1 #0) #11 (#21 #10) #20 #1030 (#40 #41)
 #51   (#74 #73 #62 #72 #71 #63 #61 #52)
 #1102 )
```

図9.

筆者のHexプログラムで連結プランを作るところまでを盤面の認識フェイズと呼んでいる．これを基にして実際に着手を選ぶ部分は，これと全く独立に使られており，プログラミングの方法もおおいに異なる．一般的な議論は第3節に譲るとして，どんな戦術を使っているか簡単に記述しておこう．なお，このプログラムは先読みを行なっていないことは特筆しておかねばなるまい．

1. 序盤（5手目ぐらいまで）は定石を参照している．これは単に強い手をうつためのみでなく，手に変化をもたせて人間が飽きるのを防ぐ意味もある．
2. 敵がすでに勝っていれば終り．
3. あと一手で自分が勝てれば（連結プランの手数が1）その手を打つ．
4. 敵のすべてのプランが未完成の台形定理（実際に石が置かれていない台形）を含み，その台形が一手で破壊できればそのように打つ．
5. 敵のプランの手数が1で，一手でそのすべてを破壊出来ればそのように打つ．
6. 自分のプランを眺めて自明でない2肢があれば，その肝要点に打つ．（もし自分のプランの手数が2ならば勝ち）
7. 敵のプランの手数，自分のプランの手数，プランの構造などに応じてピックアップされた敵の重要点のうち，自分にとっての最適点を選び出す．このとき守備を重視するか攻撃を重視するかはサジ加減一つであるが，たとえば敵の強さを示すSTRONGEなる変数の値が下ならば（敵は強いので），守備に重点が置かれる仕掛けになっている．ただ，1～6に比べ7は昔ながらの評価関数的考え方が除き切れないままになっており，ここをより精密に記述すればさらに質の高いプレイが出来るはずである．（盤面認識のフェイズはその程度のポテンシャルを秘めている）

現在Hexのプログラムはpdp11/55上のLIPQでほとんど実時間の応答をする．実力は人間の中級プレイヤーと同等程度と思われる．

2. Calculation

このゲームはトランプ一組で出来る一人遊びの中では，推理能力のいるかなり

面白いゲームである．実力の差は確かにはっきり出て，初心者の成功率が0%を含む10%以下，注意深い名人の成功率が90%以上と言われている．ルールを説明しておこう．

[目標] 4つの台に各々A,2,3,4,----,J,Q,Kなる列，2,4,6,8,---,9,J,Kなる列，3,6,9,Q,----,7,10,Kなる列および4,8,Q,3,---,5,9,Kなる列を昇順に作る．(以下上の4つの列を順に1列，2列，3列，4列と呼ぶ)

[開始・進行]

Jokerなしの52枚の札を裏向けにしてよく切る．(山)；スタックを4個確保；

loop while 山≠空；
　山から一枚めくって手に持つ；
　次のどちらかを選ぶ
　　{if 手が台の上における then おく else こちらを選べない；
　　手をスタックのどれかの上におく；
　　loop until 気がむかない；
　　　while スタックから台へ移せる；
　　　移す；

[判定] 最後に目標の台が完成していれば成功．

このゲームもGPCCの課題と出題された．筆者の作成したプログラムはHexと同様，局面の認識に重点を置いた先読みなしのものである．計算機の中で実際に表現する必要のあるのは，4つのスタックの内容と4つの台の一番上の札である．スタックの内部表現として自然なのは4×30(一本のスタックが30枚を越えないとして)くらいの二次元配列であろう．ただLIPQではこの二次元配列の処理が遅くなるので，ここではリスト構造を用いている．もともとスタックだからリスト構造を用いるのが最適なのだが，各スタックを名前(A,B,CおよびD)で参照する関係上，完全にリストデータに適合するわけではない．そこで若干の工夫が行なわれており，図10にスタック全体の構造と個々のスタック要素の構造を示してある．各スタックが両方向リストになっていることに注意しておこう．これだけではスタックの素人の網膜レベル的表現に過ぎないが，これを名人の認識，たとえば「スタックAの下から3つ目の9は2列の9，その上の4は3列の4と1便う---」のように見ることが必要である．Hexにおいて敵味方の連結プランを列挙し，それを高レベルの盤面認識として把えたのと同様，スタックの中の札を最終的にどう使うつもりかをここでの局面認識として考えようというのである．ただし，このプランにもつめがあってはならない．すなわち，実際にスタックから移し

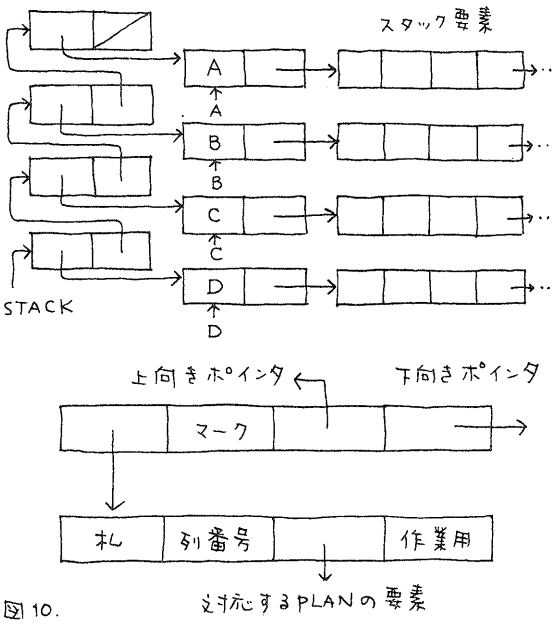
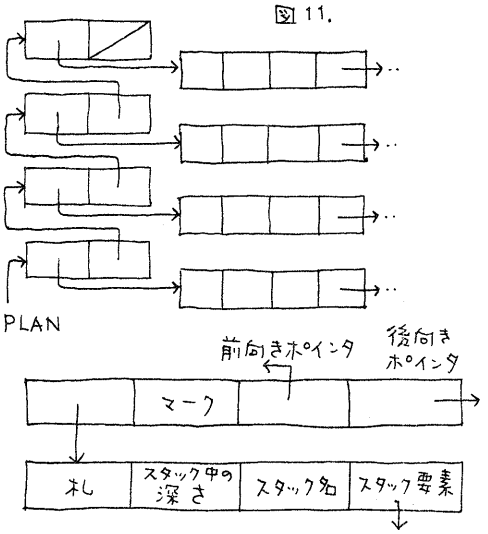


図10.

得るようなプランでなければならない．もつれのないプランを作るのはある程度アルゴリズム的な問題で，計算機にやらせても無視出来ない時間を要する．特に与えられたスタックの状態につき，もつれのないプランを総て求めようとすると先読みをシラミつぶし的にやるのと本質的に変らない手間がかかる．（筆者のプログラムも最後の手段としてそれをやることがあるがそのとたんに一手の所要時間が数十倍になる）．だから，プランを毎回最初から求めることはせずに，一度作ったプランをなるべく長く温存することを考えなければならない．新しい札をスタックにのせるとき，今自分の考えているプランが一番素直に拡張できるようなスタックを選ぶわけである．プランの構造は図11のようになっているが，通常のデバッグ用のためそれを簡単に打ち出した例を図12に示してある．（スタック，プラン共両方向リストでかつお互いにクロスリファレンスしているため，LIPQのプリティプリンティングの機能をもってしてもこのデータ構造を読みとるのは至難である．また両者がある意味で双対の関係にあるためプログラミング上混同から来る困難が非常に大きかった．）どの札をどこへ使うかについてはその安定性のレベルに応じて何段階かにランク付けがされている．たとえばKとTとQがスタック上で揃って，2列に使うことを一度決めそれ以後不都合が起らないのであれば，スタック中あるいはプラン中のそれらの札はFIXというマークがつけられる．将来不都合が起ってプランの組み替えが行なわれ得る札はMIDDLEとマークされる．あと，一時的な計算のためのマーク TEMP と，FIXされていないKを表わすHANGと計4種類のマークが現在使われている．筆者の考えではFIXとMIDDLEの間にもう一ランクあるべきだという気がする．

図11.



戦術プログラムを書く際注意しないといけないのは，プランを作成する関数やその他の補助関数がスタックにつまれた札の枚数が2の肩にのるような時間浪費のワナに陥入りやすいことで，とにかくそれを避けなければならない．筆者のプログラムが既に60%の成功率をおさめていた頃，ある勝負の終盤で成功を目前にしてこのワナに陥入って，虫ではないのに1時間以上たっても手を打ってこなかったことがあった．プログラムが完全探索の泥沼に入ったためで，それをサボるようにした途端，1秒以内でその手を打ってきた．上に述べたようなマークはプランに関する計算を速くするのに随分役立っている．またK周辺についての特別な戦術的処理を省いて，Kを他の札と同等に扱ったりすると，確かにプログラムの構造はスッキリするが，時間・成功率いずれの面から見ても性能は大幅に劣化するはずである．（筆者の実験がそれを示唆している）．これは人間だったら必

図12

```
((3 (7 6 A) 11 2 6 10 1 (5 5 A) 9 13)
 (12 2 5 8 11 1 4 (7 1 A) (10 1 B) 13)
 (4 6 8 10 12 (1 4 A) 3 (5 3 B) (7 2 B) (9 6 D) (11 2 D) (13 1 D))
 (4 (5 3 A) (6 4 B) 7 (8 2 A) (9 2 C) (10 1 C) (11 5 D) (12 4 D) (13 3 D)) )
```

らず考慮に入れているものを，プログラムの美しさなどだけを考えて省いたりすることの危険な一例である．

プログラムの戦術は大体次の7つのモデュールからなる．

1. K処理．GIVE-UP flagが立っていなければ，Kは必らずスタックDに置く．それが最後のKなら，すべてのHANG KをFIXする．そうでなければそのKにHANGマークをつける．GIVE-UP flagが下なら，Kの特別処理は行なわない．
2. 手がすぐ台札の上に置け，かつ続いてスタックから2枚以上現在のプランに従って台へ移せるならそれを実行する．
3. 手がすぐ台札の上に置け，続いてスタックから台に移せる札があれば（現在のプランに従わなくてもよい）一番多く移せるような取り方で移す．（手が台札に置ける場合，それがもつれのないプランを保証していれば必らず置くわけである．この戦術は名人の眼から見れば不十分なものである）
4. 手が9, 10, J, Qならば，HANGしているKをFIXできないかどうか調べる．出来ればそうする．その手自身もプランの中でFIXされる．
5. FIXされているプランが直接延長出来ればそうする．たとえばK, 10とFIXされていれば7がその上でFIX出来る．
6. スタックの上でプランが素直に延長出来ればそうする．たとえばスタックAのトップの8が3列の8と見做されていれば，5はその8の上に置く．
7. 手を仮にスタックの上にのせてみて（Kが全部出ておらず，GIVE-UP flagもNILならスタックDは考慮の対象外）プランを拡張してみる．その中で一番評価のよいプランが出来るようなスタックを選ぶ．プランの評価は，そのプランの入り込み具合，スタックからの移しにくさなどの要素を見て臨機応変に行なわれる．もしどのスタックに置いても今迄のプランの評価に比べてあまりに評価が悪化するようなら，すべてのMIDDLEプランを消去して，もう一度同じことを繰り返す．もしこうやってももつれのないプランが出来ないようなら，GIVE-UP flagをTにしてDの上に手をおいてみる．（Kが上に乗る危険をおかす）

ここでもHexと同様最後のモデュールは昔ながらの評価関数的発想法をつかっている．今後のプログラムの改良はなるべく最終モデュールまで行かないように途中に明確な戦術モデュールを追加することによって行なわれることになるであろう．現在のCalculationのプログラムは約700行（LISP）で，成功率は75%である．（試技100回位）成功する例で，1勝負10秒前後から5分弱まで時間消費に変化がある．ただ成功率を変えないで，時間を短縮することは可能であろう．なおこのプログラムの一人ゲーム的な使い方として，同じ手を入力してやることによる人間との実時間的な勝負がある．こうやって見てみると，このプログラムのプレイの特徴として，最後までスタックに札をため込む傾向があって，最後の2手ないし3手で一挙にスタックを掃除することが多いようである．

3. 盤面（局面）認識を中心としたゲーム・プログラミング

HexとCalculationでの実例で強調したように，盤面（局面）の戦術眼的認識は今後もっと計算機の内部表現として使われるべきである．自分の得意なゲームを自分がどのように競技しているか内観してみれば，特に早打ちチェス，早打ち碁などを考えてみれば納得出来ることかと思う．プログラムの効率の点から見ても良い内部表現の必要性はあきらかになってくる．すなわち，問題にあったデータ

構造を選ぶことがプログラミングにおいて最重要であるという常識を想い起せば良いのである．Hexの例で見たように，1手打つだけで大変な労力の要る盤面の表現をとるデメリットは，そこから戦術に必要な情報を引き出す労力が大幅に軽減されるというメリットで見事に補償される．もし生の表現から必要な情報を引き出そうとすれば，その毎々くの計算が必要となる．たとえばほとんど同じ計算を1手毎に繰り返さなければならない．これは直前までの勝負の流れを一手毎全部忘れて最初から新しい問題として考えなおすという人間では考えられないような非能率である．すなわち戦術に必要な計算を $f_1, f_2, \cdots, f_n$ とすれば，これらを1つの少し面倒な計算 g と単純な計算 $f_1', f_2', \cdots, f_n'$ を使って $f_1' \circ g, f_2' \circ g, \cdots f_n' \circ g$ という風になんとか分解しなくてはならない．こうすればgを1回計算すれば事足りるからである．これはゲーム・プログラミングに限らず効率を重んずるところでは常に必要なことである．

では盤面の戦術眼的認識に対応する内部表現をどうやって見付けるか．このためには通常のゲームの木の探索法によるゲーム・プログラミングを一回忘れてみて先読みをしないプログラム作りを真面目に検討してみるのが一番である．こうすればアイデアが自ずと浮かぶ．何故なら先読みをしないという制限条件のもとでは，本来先読みによって得られるような情報をパターン認識的な発想で求めざるを得なくなるからである．言い換えれば，早碁打ちがやるように，良い盤面の表現は先読みで得られる情報が影のように焼きついたものなのである．簡単な例だが，Hexにおける支持節点つきの連結情報は，ルール記述のレベルから言えば先読みからでしか得られない情報である．先読みのことを考えるのは，これらの考察の後で十分である．（良い盤面認識は先読みの経済化にも役立つ．ここではまだ実際例がないのでその議論は省略する）

盤面の表現を上に述べたような方法でリファインしていく上でもちろんいくつかの問題点はある．Hexの例で見たように，グラフ表現として盤面がまずあり，その上に連結プランとして表現される高次の盤面認識があったわけだが，このような積み重ねの中で大きな戦術眼の欠落が起こり得る．たとえば連結プランとして最小手数のものしか考えないのは，プランから外れていてしかも重要な石を見落すことになる．実際，現在のプログラムは攻撃的な手を打つ際，石や直接的に過ぎる感がある．またより大きな盤面では石の厚みとか支配というような連結プランよりもさらに高次の認識を必要になるわけで，本当に強いプログラムにするには盤面認識の階層構造をさらに高くしなければならない．Calculationの例でいえば，名人のレベルで非常に重要なこととされているプランの融通性，つまりその後出て来る札の出方に柔軟に対応出来るようなスタックの構成は，ある意味でプランの集合を"プラン"として持つことである．つまりこれはちょうどHexにおける第1階の2股と第2階の2股の関係であり，プランを一つ探すのに苦労している段階ではそれより一階上のプランを求めることはより大変になる．これと同様の状況は先読みにおいても起る．先読みの目標が第n階のプランに関するものであるとすると，そこでは自然に第n+1階の話をしていることになるのである．今迄の議論から推察出来るように，盤面認識のことをきちんと考えたプログラムは一般に終盤に強い．そして盤面認識の階層構造を高くしたものほど中盤から序盤にも強くなる．これは人間の初心者がゲームに強くなっていく段階に非常に似ており，普通の先読みと評価関数を使ったプログラムが初盤・中盤にかけてはなんとかいい手を打つが，終盤では極端に悪くなる（例が多い）のと対照的である．

最後に上記のようなゲーム・プログラミングに特徴的なことを一つ注意しておこう。それは戦術プログラムがつぎはぎだらけのプログラムになる，あるいはなるべきことである。逆に言えば最初からすっきりと設計出来るような戦術プログラムを持つゲームは，もともとtrivialか，少なくとも人間の思考過程の研究としては興味のうすいものなのである。戦術プログラムは実戦を重ねていくと，必らず欠点が見つかるものであって，あれこれ修正しているうちに膨大になり全体の構造が不透明になる。これも人間がゲームに強くなる過程によく似ている。だから真に不透明なプログラムになって始めて強いプログラムが出来たと喜ぶべきであって，いわゆるstructured programming的な発想は害になりこそはすれ，益にはならないと思う。

戦術プログラムには，プログラム上の虫と戦術上の虫の両方が混在している。今迄述べたような方法をとれば，従来の方法に比べ戦術上の虫取りは極めて人間的でやりやすい。何故なら，相対的に盤面認識の論理が深いので，着手選択の論理自身は割合浅いからである。すなわち悪手に対する因果関係が割合はっきりしている。膨大な先読みの木をトレースしてみる必要はさらさらない。戦術のデバッグのためにいくつかのユーティリティを最初からプログラムに組込んでおくのは賢明である（どうせ最初から完璧なプレイなど出来ないのだから）筆者の場合，HexやCalculationの悪手をピックアップしてその原因を直すのに非常に簡単なオンライン的インスペクションで済んだ。いずれにせよ，ゲーム・プログラミングにはよい会話型のシステムが必要不可欠であろう。

おわりに

筆者は今度長年の夢だった囲碁（11×11ぐらいで）に取り組んでみたいと思っている。そのときはここで述べた議論をさらに具体的に推し進めることが出来ると思う。

謝辞

日頃御指導いただく池野信一室長，いろいろ御協力いただいた若葉悠調査員，奥乃博主任に感謝します。毎度有難とうございました。