An Approach to Conquer Difficulties in Developing a Go Playing Program

YOSHIHISA MANO*

In this paper, we show an approach to developing a Go playing program. First, we describe the method to represent and implement the situation on a Go board. We formally define the operations for objects with a hierarchical structure, and implement them by applying the localization and abstraction techniques to our Pascal program. Thus we have a modularized program in which the realization parts and utilization parts are separated only with a few interfaces. Second, we describe the usage of the Go pattern knowledge, and a human oriented non-procedural language Gopal. A pattern in a Gopal program consists of the conditions to characterize it and the enumeration of move candidates. We have been convinced of its descriptive capability and understandability by writing many patterns.

1. Introduction

The game of Go is a strategic game, and we can find analogies between the decision processes of playing a Go move and the decision processes an individual or organization uses to establish a policy. Although the efficient game tree search is successfully used in many chess programs, it is considered necessary for Go playing programs to simulate the human's decision processes. This includes the application of a player's knowledge, the analysis, the inference, and planning because of the complexity which is inherent in the game of Go. We can say that this game is a good subject for research on human decision processes[1].

First of all, a Go playing program must represent and recognize the situation on a Go board so that it can estimate the state of things and build its plan. In order to recognize the situation, it must represent the various kinds of abstraction levels with complex relationships as human players do.

Among the Go playing programs, the INTERIM.2[2] ranks best. It has some interesting data structures which are models of human recognition methods, such as lenses to monitor patterns, and webs to examine the specific local status. It is important to be able to build such information in the program systematically, since the size of the complete program will become very large.

Go is also a visual game. One of the essential reasons why humans can become good players is that they can accumulate considerable knowledge on general and abstract Go patterns, and have an efficient pattern matching mechanism.

The Go program by Zobrist[3] uses templates to represent such pattern knowledge, and has the ability to scan templates on the board and to enumerate move candidates with their weights. However the knowledge is simple and inflexible and has less extensibility. It seems difficult for human designers to modify or extend the set of templates. As human players become experts by learning something from their games, the Go programs should be improved through their experiments.

In this paper we will describe, from the viewpoint of software engineering, an approach to the first stage of our project [4, 5, 6] to construct a Go playing program.

2. Implementation of the Representation of a Go Situation

2.1 Representation of a Go Situation

Let a set of **points** P be $\{(x, y) | 1 \leqq x \leqq 19,\ 1 \leqq y \leqq 19\}$ and a set of colours C be {Black, White, Empty}. Then the configuration in the Go board can be represented by a configuration function $\sigma: P \rightarrow C$*1. For $N \subset P$, we can construct a graph N^* whose nodes are the elements of N. The edge between $(x1, y1)$ and $(x2, y2)$ exists if $|x1-x2|+|y1-y2|=1$. $\sigma^{-1}(c)^*$, where c is either Black or White, is composed of some connected subgraphs, and we call a set of the points in each connected subgraph a **string** of the colour c.

From the standpoint of formalizing or formulating the Go rules or the animation of stones, it is necessary to regard strings as the basis of the game Go. From the standpoint of implementing Go playing programs, however, the concepts of a higher level are necessary.

We have taken into account the ability to recognize the situation from both the top-down and bottom-up viewpoints for making a plan. And we have defined a set of types of **objects** as elements of the board configuration. Each object has a set of attributes, which the type provides.

*Computer Science Division, Electrotechnical Laboratory, 1-1-4 Umezono, Sakura-mura, Niihari-gun, Ibaraki, Japan.

*1For representing the progress of the game, it is necessary to include such information as the next turn, the inhibited point because of 'Ko' and the number of captured stones. Since such information can be easily built, we will not mention it again.

The types of levels higher than point and string cannot be defined clearly. Human players cannot be clearly defined either. Our situation is that we prepare some concepts for the estimation of configuration and for strategic planning, and provide the program with the actual definition for those concepts. Thus, as the human players refine their recognition of the board situation in proportion to the improvement of their skill, our Go program will refine the actual definition of each type in proportion to its skill. We think that the change in recognition because of the improvement of the program's skill can be regarded as the change in the number and quality of attributes. The structure of those objects, therefore, can be fixed.

The informal definitions of the objects except point and string are as follows.

group: This is a set of strings of the same colour, whose any two elements can be connected into a string whenever the enemy will attack*[2].

family: This is a set of groups of the same colour, and it is necessary to have a strategy under the assumption that any two elements will be combined into a group.

linkage: This is the supposed line between two points of the same colour or one between a point with a stone and its nearest edge with a distance less than four, which can be connected whenever the enemy will attack*[2].

loose linkage: This is the supposed line between two points of the same colour or one between a point with a stone and its nearest edge with a distance less than four, and it is profitable to have a strategy under the assumption that its any two elements will be combined into a group.

string relation: This is the relation of two strings with the same or opposite colours.

group relation: This is the relation of two groups with the same or opposite colours.

family relation: This is the relation of two families with the same or opposite colours.

The main roles and the attributes of the Go objects are shown in Table 1.

There are many problems in implementing such representation. They are, for example:

(1) to efficiently process the dynamic creation and deletion of objects in a large amount of data,
(2) the same problem as (1) for the hypothetical configuration during a lookahead sequence,
(3) to easily append and refine the set of attributes for improving the program.

Although they are important factors, they make the program complicated, and should be separated from the inherent complexity of the Go game.

*[2]More precisely, the attack is limited to one which will not bring any damage to the enemy himself.

Table 1 The Go objects.

Objects	Main Roles	Main Attributes
point	board configuration, influence from/to other objects	colour, string name potential value,
string	a unit which is alive or dead simultaneously	number of stones, 'Dame' points, vitality
group	a unit of battle, a tactical base	size, extension direction, vitality, opposite groups
family	a strategic base, a unit for situation estimation	territory size, local-tactics for invading
linkage	to surround territories, an object to attack and defend	type of linkage, length, strength, weak points
loose linkage	to surround potential territory ('Moyo')	type of loose linkage, length, strength, weak points
string relation	to keep information on hostility or cooperation	common 'Dame' points, contact points
group relation	to keep information on hostility or cooperation	common enemy group, local-tactics on battle
family relation	to keep information on hostility or cooperation	contact area, local-tactics on contact area

For coping with such problems, one of the most valuable rules in software engineering teaches us to divide the problem into abstraction levels and to construct the program in a hierarchical structure depending on the structure of abstraction levels[9, 10]. Elements such as data types and operations implemented in the lower abstraction levels are used by modules in the upper abstraction levels based on only their specification. Since the interface between them can be small and formal, we can easily validate or prove the correctness of the program.

We have considered the representation of the situation on a Go board as the lowest abstraction level, and we have decided to construct the main part of the Go program above it, strictly applying that programming methodology. In other words, we have first formally defined the operations for creating, deleting, merging and scanning the objects, and other operations for accessing and modifying the attributes of objects. Thus we only use the specification when implementing both the main part of the Go program and the operations on the data structures.

Each object is modelled by a tuple with a set of attributes. The size and structure of the tuple and the set of attributes are determined by the type of the object. Components of a tuple which represent the hierarchical structure may be other objects. The structure of tuples and attributes used in the specification is shown in Table 2. In this table, for example, an object of type family is a 2-tuple composed of a set of groups

Table 2 The structure of Go objects.

$C=(\{F_1, F_2, \ldots\}, \{L_1, L_2, \ldots\}, \{LL_1, LL_2, \ldots\})$
  Attr(C)= {*Colour*: (Black, White); . . .}
$F=(\{G_1, G_2, \ldots\}, \{FR_1, FR_2, \ldots\})$
  Attr(F)= {*Colour*: (Black, White);. . .}
$G=(\{S_1, S_2, \ldots\}, \{GR_1, GR_2, \ldots\})$
  Attr(G)= {*Colour*: (Black, White); . . .}
$S=(\{P_1, P_2, \ldots\}, \{SR_1, SR_2, \ldots\})$
  Attr(S)= {*Colour*: (Black, White); . . .}
$P=(\ )$
  Attr(P)= {*Coordinate*: integer × integer,
    State: (Black, White, Empty); . . .}
$L=(\{P_1, P_2\})$
  Attr(L)= {*Colour*: (Black, White); . . .}
$LL=(\{P_1, P_2\})$
  Attr(LL)= {*Colour*: (Black, White); . . .}
$FR=(\{F_1, F_2\})$
  Attr(FR)= { ; . . .}
$GR=(\{G_1, G_2\})$
  Attr(GR)= { ; . . .}
$SR=(\{S_1, S_2\})$
  Attr(SR)= { ; . . .}
where
  C: Player, L: Linkage,
  P: Point, LL: Loose Linkage,
  F: Family, FR: Family Relation,
  G: Group, GR: Group Relation,
  S: String, SR: String Relation.

in the object and a set of family relations, and one of its attributes is *Colour* which shows the colour of the family.

The specification we have defined in this mathematical model is a set of relations of states, before and after each operation [11]. The specification of operations on object family is shown in Table 3, where X/i is the i-th component of the tuple for object X, and ' means the value after the operation. The 'assume' part shows the condition for applying the operation, and the 'effect' part describes the effects of the operation in the form of the relations of the states before and after each operation. This table shows, for example, that when two families merge ('*MergeFamily*'), new family F_3 is created, which replaces F_1 and F_2, and whose first and second component is $F_1/1 \cup F_2/1$ and $F_1/2 \cup F_2/2 - \{FR|FR/1=\{F_1, F_2\}\}$ respectively, where $\cup$ and $-$ are the operators union and difference of sets respectively. The specification does not mention most attributes, since they are to be defined in the upper abstraction levels.

2.2 Implementation of the Operations

The formal specification uses the mathematical model on sets and tuples, and is simple and understandable as the interface between two abstraction levels.

We have selected Pascal as the implementation language. Pascal is a structured language, but has several inconvenient points when using the implementation method based on the above specification. In this section we will describe the problem areas and our solutions.

First of all, according to the methodology, the set of

Table 3 Description of basic operations on Families using the states machine model.

where
  ϕ means an empty set,
  each of C, C_b and C_w is an element of Players,
  each of F, F_1, F_2 and F_3 is an element of Families, and
  NewXxx (*InitVal*) is a generator of the object
    of *Xxx* type with the initial value *InitVal*.

operation *Initialize* ( )→{C_b, C_w}
 effect
  C_b=*NewPlayer* ((ϕ, ϕ, ϕ)),
  C_w=*NewPlayer* ((ϕ, ϕ, ϕ)),
  Attr(C_b, *Colour*)=Black,
  Attr(C_w, *Colour*)=White.
operation *CreateFamily* (C)→F
 effect
  F=*NewFamily* ((ϕ, ϕ)),
  $C/1'=C/1 \cup \{F\}$,
  Attr(F, *Colour*)=Attr(C, *Colour*).
operation *Families* (C)→{F_1, F_2, . . .}
 value $C/1$.
operation *PurgeFamily* (F)→
 let C such that $C/1$ includes F
 effect $C/1'=C/1-\{F\}$
 assume $F=(\phi, \phi)$.
operation *MergeFamily* (F_1, F_2)→F_3
 let C such that $C/1$ includes F_1 and F_2
 effect
  F_3=*NewFamily* (($F_3/1$, $F_3/2$)),
  where
   $F_3/1=F_1/1 \cup F_2/1$
   $F_3/2=F_1/2 \cup F_2/2-\{FR|FR/1=\{F_1, F_2\}\}$,
    ('F_1' and 'F_2' appeared in $F_3/2$ are renamed by 'F_3')
  $C/1'=(C/1-\{F_1, F_2\}) \cup \{F_3\}$.
 assume
  Attr(F_1, *Colour*)=Attr(F_2, *Colour*).
operation *ReferFamilyAttributes* (F)→
    Access-method to Attributes of F for referring
 effect none.
operation *ModifyFamilyAttributes* (F)→
    Access-method to Attributes of F for modifying
 effect none.
operation *ExcludeGroup* (G)→
 let F such that $F/1$ includes G
 effect $F/1'=F/1-\{G\}$.
operation *IncludeGroup* (F, G)→
 effect $F/1'=F/1 \cup \{G\}$
 assume
  for each F_i, $F_i/1$ does not include G,
  Attr(F, *Colour*)=Attr(G, *Colour*).

routines which pertain to the above formal specification should be implemented in the data management module, and the details such as internal data structures and algorithms should be hidden from external routines. In fact this data management module uses several techniques in respect to the management of data and memory. The details which should be hidden include:

(1) the method to give a unique identifier to each object,
(2) the fact that there is an array to monitor all objects, and that identifiers are used as indices to access the array elements,
(3) the method of always using the array when

tracing some relating objects, in order to support the data sharing mechanism described in (5),

(4) the method of managing the free memory of deleted objects,

(5) the way of realizing the effective use of space and time by sharing as many common data as possible for hypothetical configurations during a look-ahead sequence.

In order to implement the data management module according to the discipline described above, it is necessary to use some language features for controlling the scope of names or for describing abstract data, such as the module facility in Modula-2[13]. Using such language features, it is possible to make some routines accessible from others while prohibiting the access to the internal data common to those routines. In our Pascal system[14] which has a module facility similar to a Modula-2 module, we can set the special scope of visibility of names such as variables and routines. Thus, a set of routines can be implemented in the data management module corresponding to the operations in the formal specification, except that functions to extract a set of objects such as 'Families' in Table 3 are realized by pairs of functions.

On the other hand, the access to attributes, not for the data management module but for the upper level modules, is directly done through a pointer to an attributes record. These pointers are obtained by invoking the functions in the data management module. Two such functions are provided for each object in order to distinguish between the access for referring and the access for modifying. We can append and refine the set of such attributes independently of the data management module, except that it is necessary to recompile the module. We can say, therefore, that our implementation of the use of these attributes is the abstraction of the procedures for accessing data. Furthermore, the access of attributes is efficient because we need only one invocation of a function to obtain the pointer to the record.

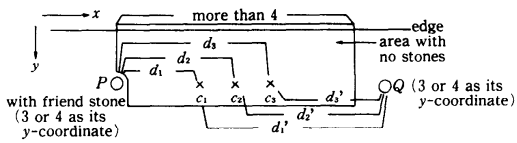
The methodology which uses the area for dynamic data as a data base is applicative and a very useful method for obtaining structured programs of various applications which treat dynamic data. Access to the data base is abstracted by using the pointer-valued functions.

3. Non-procedural Description of Go Pattern Knowledge

3.1 Go Pattern Knowledge

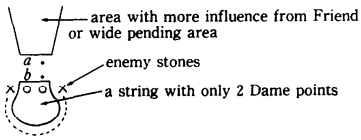
One of the main features of humans' game tree search is that the tree is narrow and deep but the result is nevertheless accurate. In addition to the fact that humans have excellent ability for accurate estimation of a situation, we must take into account humans' generalized and abstracted Go pattern knowledge and

(a) Extension from a Friend stone



where
$d_1+d_1' \geqq 6$, $d_1=3$, $d_2 \fallingdotseq d_2'$, $d_3'=3$.
C_1 (3 or the y-coordinate of P as its y-coordinate),
C_2 (3 or the y-coordinate of P or Q as its y-coordinate) and
C_3 (3 or the y-coordinate of Q as its y-coordinate).
V_p (V_q): the vitality of the group including $P(Q)$,
whose value is one of 1, 2, 3 and 4
(the larger, the more vitality).
if Enemy stone on Q then
case $V_p < V_q$: C_1, C_2 are candidates,
case $V_p = V_q$: C_2, C_3 are candidates,
case $V_p > V_q$: C_3 is a candidate.
if Friend stone on Q then
case $V_p < V_q$: C_1 is a candidate,
case $V_p = V_q$: C_1 and C_2 are candidates,
(Note that the candidate C_3 for $V_p = V_q$, and the candidates for $V_p > V_q$ are unnecessary because of its symmetry.)

(b) Escape of a string with 2 Dame points



if the string is not captured by Sicho after point a of Friend move and then point b of Enemy move, point a can be a candidate.

Fig. 1 Examples of Go pattern knowledge.

efficient pattern matching ability. Thus humans can find a small set of candidates in which the best move exists for the most cases. The generalized and abstracted pattern knowledge could be such knowledge as 1) symmetric patterns are memorized collectively or not memorized (derivable through some transformation from some basic forms), 2) the way it is on the local field, 3) the relations with the surroundings are described in abstract terms, and 4) each pattern knowledge has not a concrete configuration but a set of configurations satisfying certain conditions.

For example, Fig. 1 shows the illustration of two examples of such pattern knowledge*[3]. Fig. 1(a) is a knowledge of the candidates for the next move on 'Extension' from a friend stone toward a wide side area where any moves are not yet played. In accordance with the surrounding conditions, C_1, C_2 or C_3 are the potential candidates. Fig. 1(b) is a knowledge of 'Escape' of a string with only two consecutive 'Dame (liberty)' points.

It will be possible to simulate the human decision

*[3]These two pieces of knowledge are beginners' ones. The higher the rank of the player, the more cases and the more sophisticated conditions on the surroundings he understands.

processes, if the program has many pieces of such pattern knowledge and uses them as the resources to construct its lookahead sequences.

3.2 Non-procedural Language for Go Pattern Knowledge

We made a decision to utilize such pattern knowledge in our Go program in order to enumerate move candidates, and we have investigated that those patterns can be characterized by the objects and their attributes described in Section 2. Since the structure of such knowledge is not sufficiently analyzed yet, it is much more important for us to be able to modify, append or exchange a set of the pattern knowledge easily.

It is required, therefore, that the description language for such knowledge is declarative or non-procedural, and supports the systematic and abstract description for related, symmetric or similar patterns with some common features. In addition to the above requirements, the descriptive power and the executive performance is expected to be high.

Since any existing language does not satisfy such demands, we have designed and implemented a language for Go pattern knowledge, called Gopal. We have since written and accumulated such pattern knowledge in Gopal. Each pattern knowledge written in Gopal has two parts. One is for conditions to characterize the pattern, and the other is for enumerating the move candidates. A Gopal program, which has a set of pattern knowledge, acts like a production system. However the different order of pattern applications does not cause different results since each application has no side effects.

As conditions, the following should be described:

(1) the time to be applied (such as an initial stage, 'Semeai' (the mutual attack) and when making eyes), the place to be applied (such as the corner and the side), and the objects to be applied (such as a stone, a string and a linkage),
(2) a configuration of a local field, allowing the indefinite states,
(3) the existence of objects and the relations between their attributes,
(4) the same conditions as (3) for the hypothetical configuration after a predeclared lookahead sequence.

As the declaration of move candidates, we should describe not only the candidates themselves but also some data, which is useful for constructing the local-tactics such as:

(1) the reason why the moves are declared, and
(2) some numerical values whose meaning is defined for each reason.

3.3 The Language Specification of Gopal

Fig. 2 is an example of a Gopal program which represents the two kinds of pattern knowledge in Fig. 1. This subsection gives the main design decisions and a brief explanation of the main constructs of Gopal, referring to the example program in Fig. 2 when necessary. The readers will be able to see the natural correspondence between the human-oriented representation in Fig. 1 and the Gopal program in Fig. 2.

The amount of pattern knowledge which human players have is probably large. The pattern knowledge should be modularized according to the relationship or the amount of common parts. A set of pattern knowledge is described in some Gopal programs, and a Gopal program ('Example', in Fig. 2) consists of some descriptions of pattern knowledge ('ExtToBoth' and 'Escape2', in Fig. 2) which should have close relations to each other ('ExtToBoth' and 'Escape2' in Fig. 2, however, have little relations). As mentioned later, some systematic pattern description can be allowed in one pattern description.

A Gopal program can have some parameters which are used typically to specify the particular local battlefield where some of the patterns in the program will be

```
GopalProgram Example;
applied_time  AnyTime;

pattern ExtToBoth;
(* Extension from Friend toward Friend or Enemy *)
 applied_time  Joban;
 applied_place  Side;
 object  P,Q, C1A,C1B,C2A,C2B,C2C,C3B,C3C: Point;
         C1X,C2X,C3X: BoardIndex;  Vp,Vq: integer;
 pickup  P in Friend;
 condition
     P.y in [3,4]
   & P.x<=BoardSize-9
   & Emp1Area(PNT(P.x,1),5,3,P)
   let Q=Near(P,Range(PNT(P.x+6,1),10,4),Both) end_let
   & Q.y in [3,4];
 candidate
   let C1X=P.x+3; C2X=(P.x+Q.x+1) div 2; C3X=Q.x-3;
       C1A=PNT(C1X,P.y); C1B=PNT(C1X,3);
       C2A=PNT(C2X,P.y); C2B=PNT(C2X,3); C2C=PNT(C2X,Q.y);
                         C3B=PNT(C3X,3); C3C=PNT(C3X,Q.y);
       Vp=Vitality(GrpName(P)); Vq=Vitality(GrpName(Q)) end_let
   subpattern;
     condition  State(Q)=Enemy;
     candidate
       subpattern;
         condition  Vp < Vq;
         candidate  with (Kakucho,65.0,1) C1A,C1B, C2A,C2B,C2C;
       end_subpattern;
       subpattern;
         condition  Vp = Vq;
         candidate  with (Kakucho,65.0,1) C2A,C2B,C2C, C3B,C3C;
       end_subpattern;
       subpattern;
         condition  Vp > Vq;
         candidate  with (Kakucho,65.0,1) C3B,C3C;
       end_subpattern;
   end_subpattern;
   subpattern;
     condition  State(Q)=Friend;
     candidate
       subpattern;
         condition  Vp < Vq;
         candidate  with (Kakucho,65.0,2) C1A,C1B;
       end_subpattern;
       subpattern;
         condition  Vp = Vq;
         candidate  with (Kakucho,65.0,2) C1A,C1B, C2A,C2B,C2C;
       end_subpattern;
   end_subpattern;
end_pattern;

pattern Escape2;
(* Escape of a string with 2 Dame points *)
 applied_place  AnyPlace;
 object  P: Point; FS: StringId;
         IsSicho: Boolean; Pot,Dens: real;
 pickup  P suchthat Match(1,'O..+/
                             O..+');
 condition
   let FS=StrName(P); Pot=Potential(PNT(P.x+3,P.y));
       Dens=Density(PNT(P.x+3,P.y)) end_let
     NoDame(FS)=2
   & ((Pot>-0.2) and (Pot<0.5)) or (Dens<0.2)
   modifying [StrData] supposing PNT(P.x+2,P.y),PNT(P.x+1,P.y)
     let IsSicho=EmpPSet(SichoE(FS)) end_let
   & not IsSicho;
 candidate  with (Nige,30.0,FS) PNT(P.x+2,P.y);
end_pattern;

end.
```

Fig. 2 An example of Gopal program.

applied, although this facility is not used in Fig. 2.

For all or a part of descriptions of pattern knowledge in one Gopal program, the time and the place when and where they are applied are explicitly designated. It is also explicitly designated whether each pattern description should treat the symmetric patterns (4 patterns for only the rotations, or 8 patterns for the rotations and the reflections). They are written in parts of **applied_time** and **applied_place**.

The names, which are bound with values or objects such as of integers or strings, are declared in the **object** declarations. Each name must be bound just one time in order to avoid the introduction of the concept 'order of execution'. The names, which are bound in **for_each** phrase or **for_some** phrase described below, are only exceptions. The translate-time constant names may be bound in their declarations for the simplicity of descriptions and the possibility of optimization.

A description of pattern knowledge is mainly composed of a **condition** part in which a relational expression to characterize the pattern is written, and a **candidate** part in which some move candidates for the local fields satisfying the relation are listed.

A relational expression to characterize the pattern is a logical expression, and several built-in data types and many built-in functions are provided in Gopal in order to increase the descriptive power. For example, 'Colours', 'Point' and 'PointSet' are built-in data types of Gopal, and many Gopal functions whose values are of Colours type, Point type, PointSet type and so on are also provided.

In Fig. 2, for example, a built-in function PNT(x, y) returns a Point type value with the given coordinates; a Boolean function EmplArea(P, dx, dy, Q) returns true if there are no points with stones except the point Q in the rectangle defined by two points P and ($P.x+dx$, $P.y+dy$)*4; a Point type function Near(P, R, C) returns a point with a stone of colour C, which is in the set of points R and is the nearest to the point P; a PointSet type function Range(P, dx, dy) returns a set of points in a rectangle defined by a given point and a given size; GrpName(P) returns the identifier of the group which includes the point P; and Vitality(X) returns an approximate value of the vitality of an object X (the accuracy is not defined in Gopal).

It is often necessary to examine whether some conditions are satisfied for each point or for some point in the data of PointSet type. Gopal has such facilities in the form of **for_each** and **for_some** phrases respectively (the syntax is: **"for_some"** (or **"for_each"**) name **"in"** PointSet-type-expression "(" logical-expression ")")*5. The name after **for_each** or **for_some** is bound with each point in the data of PointSet type in the phrase, but its value is undefined outside of the phrase.

The syntax of Gopal logical expressions is similar to those of Pascal with an additional operator '&', and '&' has the same meaning as **and** except for the efficiency.

The candidates, which are the results of application of the pattern knowledge, are written in **with** phrases (the syntax is: **"with"** "(" intention "," a-sequence-of-numerical-data ")" a-sequence-of-candidates) with an intention, a weight, and some numerical data whose meaning is defined for each intention. The candidates are those for local battlefields, and the modules of higher levels which have some global plans decide which one of the candidates should be moved in the real match.

There may be various derivative patterns with additional conditions from a basic pattern. In Gopal, **subpattern** parts, in which there exist a **condition** part and a **candidate** part, and which may be nested, are provided in order to write such derivative patterns systematically.

The names declared in **object** declarations are used to be bound with a value and to refer to that value elsewhere. To bind a name with a value, a **let** phrase is used (the syntax is: **"let"** name "=" expression ";". . . **"end_let"**). As described above, the binding for each name occurs only once in the text. A **let** phrase may be placed rather freely in the condition part and candidate part.

Human Go players often examine some values or relations in some hypothetical configurations after rather straightforward lookahead sequences predictable for the pattern. In order to support this tactic, Gopal has **supposing** phrases (the syntax is: **"modifying"** information-class **"supposing"** lookahead-sequence **let** phrase). After the word **modifying**, the class of necessary information in the hypothetical configuration is specified for the efficient lookahead. The **supposing** phrase in Fig. 2 is one to certify in the hypothetical configuration, that after two moves it will not be captured by 'Sicho'.

It must take much time to search the patterns in the configuration which satisfy given conditions. A **pickup** phrase (the syntax is: **"pickup"** name **"in"** object, or **"pickup"** name **"suchthat"** logical-expression) specifies a particular object, which is used at the first stage of the pattern matching. By restricting the range to be searched for pattern matching, the process can be done efficiently. A special function 'match', which is used only in **pickup** part, gives a specific local configuration, where 'O' in a character string means a point with a friend stone, 'X' an enemy, '.' a vacant, '*' any point, '+' a friend or a vacant, '−' an enemy or a vacant, '/' the end of a row, and so on. The first parameter 1 of 'match' in Fig. 2 means that the first point in the string is named P.

3.4 Gopal Implementation

A Gopal program is translated into a Pascal external program (a compile unit which is invoked from others) by a Gopal translator. The Pascal programs are compiled using a common environment file. This environment file includes the definitions of the constant names

*4 $P.x$ and $P.y$ means the x and y coordinates of a point P respectively.

*5 "a" means the terminal symbol a.

and the type names which are commonly used, and this file will be linked together with other modules such as Go main module, the data management module and the Gopal library. The Gopal translator is a one-pass recursive descent translator written in Pascal.

The whole structure of our Go program is described in Fig. 3, including the data management module and not yet implemented parts.

Fig. 4 shows a part of the names of pattern knowledge descriptions which are implemented in Gopal, and an example of their execution. Fig. 4(a) is a brief explanation of the patterns, Fig. 4(b) is a Go board configuration and shows the potential candidates after an execution of the Gopal programs, and Fig. 4(c) is the trace of an execution of Gopal programs. In the Fig. 4(c), the number in 'pick' field is the number of times when the pickup condition is satisfied, the number in 'cand.' field is the number of candidates which the pattern has listed, and the number in 'msec' field is the CPU time in milliseconds.

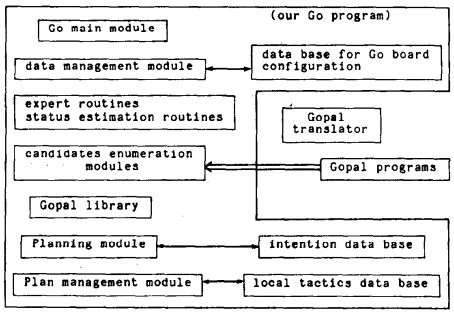


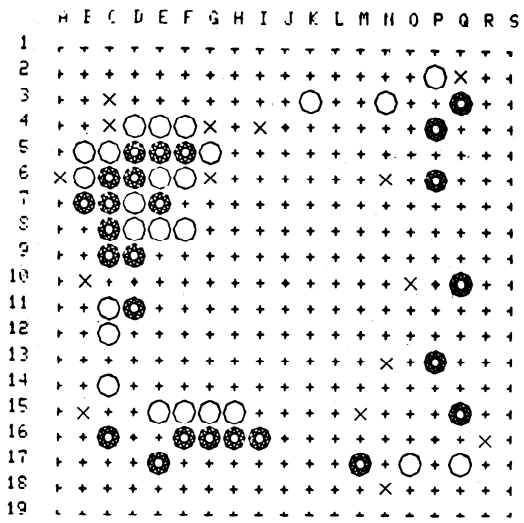
Fig. 3 The structure of our Go program.

(a) patterns in Gopal programs

```
=====   GopalProgram Fuseki;  =====
pattern AkisumiSen; : First move to empty corner
pattern HShimari;   : Shimari from Hoshi
pattern KShimari;   : Shimari from Komoku
pattern SShimari;   : Shimari from 3-3
pattern HKakari;    : Kakari to Hoshi
pattern KKakari;    : Kakari to Komoku
pattern SKakari;    : Kakari to 3-3
pattern Ext3;       : Extension from 3rd line
pattern Ext4;       : Extension from 4th line
pattern ExtB;       : Extension from Friend
pattern ExtE;       : Tsume toward Enemy
pattern ExtU;       : Upward extension
pattern Inv33;      : Uchikomi of 3-3
pattern InvMiddle;  : Uchikomi
=====   GopalProgram Battle;  =====
pattern Capture;    : Capturing Enemy string
pattern UnCapture;  : Escape of threatened string
pattern Threat;     : Ate
pattern Tsugi1;     : Tsugi of Kosumi
pattern Tsugi2;     : Tsugi of Ikken
pattern Tsugi3;     : Tsugi of Niken
pattern Cut1;       : Kiri of Kosumi
pattern Cut2;       : Kiri of Ikken
pattern Cut3;       : Kiri of Niken
pattern PostTsuke1; : Coping with downward Tsuke
pattern PostTsuke2; : Coping with horizontal Tsuke
pattern PostTsuke3; : Coping with upward Tsuke
pattern Kake1;      : Kake to a string with 2 Dames
pattern Kake2;      : Kake to a string with 3 Dames
pattern Esc1;       : Escape of a string with 2 Dames -1
pattern Esc2;       : Escape of a string with 2 Dames -2
pattern EscU;       : Upward Escape
pattern SumiAttack; : Attack to a corner
=====   GopalProgram Yose;  =====
pattern Suberi;     : Invading by Suberi
pattern HaneYose;   : Yose by Hane
pattern KosumiYose; : Yose by Kosumi
pattern OsaeYose;   : Yose by Osae
  :
  :
```

(b) A Go board configuration

×: Black's move candidate



(c) An example of the execution of the Gopal patterns for the Go configuration (b)

```
(     Fuseki)
AkiSumiSen/    4 pick/  0 cand./    16 msec
Ext3       /  74 pick/  1 cand./    87 msec
Ext4       /  74 pick/  0 cand./    58 msec
ExtB       /  74 pick/  0 cand./    34 msec
ExtE       /  76 pick/  0 cand./    54 msec
ExtU       /  74 pick/  5 cand./    86 msec
HKakari    /   8 pick/  0 cand./    20 msec
HShimari   /   8 pick/  0 cand./    24 msec
Inv33      /   4 pick/  1 cand./     0 msec
InvMiddle /  76 pick/  1 cand./    59 msec
KKakari    /   8 pick/  0 cand./    14 msec
KShimari   /   8 pick/  0 cand./    30 msec
SKakari    /   8 pick/  0 cand./    16 msec
SShimari   /   8 pick/  0 cand./     0 msec
           <-----   672 msec
(     Battle)
Capture    /  13 pick/  0 cand./    19 msec
Cut1       /   3 pick/  3 cand./    95 msec
Cut2       /   1 pick/  0 cand./    37 msec
Cut3       /   2 pick/  0 cand./    51 msec
Esc1       /  37 pick/  0 cand./    39 msec
Esc2       /   3 pick/  0 cand./    54 msec
EscU       /  37 pick/  0 cand./    42 msec
Kake1      /   1 pick/  0 cand./    76 msec
Kake2      /   0 pick/  0 cand./    59 msec
PostTsuke1/    0 pick/  0 cand./    49 msec
PostTsuke2/    0 pick/  0 cand./    44 msec
PostTsuke3/    0 pick/  0 cand./    33 msec
SumiATTACK/    4 pick/  0 cand./    10 msec
Threat     /  13 pick/  0 cand./    53 msec
Tsugi1     /   0 pick/  0 cand./    35 msec
Tsugi2     /   0 pick/  0 cand./    44 msec
Tsugi3     /   0 pick/  0 cand./    78 msec
UNCapture /  13 pick/  0 cand./    20 msec
           <-----   1095 msec
(     Yoseru)
HaneYose   /  74 pick/  1 cand./    42 msec
KosumiYose/   74 pick/  4 cand./    38 msec
OsaeYose   /   1 pick/  1 cand./    57 msec
Suberi     /  74 pick/  0 cand./    55 msec
           <-----   246 msec
  :
  :
cand. intention weight      values        pattern
 P8  (  Kakucho, 60.0    1    0    0    0) /Ext3
M15  (  Kakucho, 58.0    2    0    0    0) /ExtU
N13  (  Kakucho, 50.0    2    0    0    0) /ExtU
 N6  (  Kakucho, 58.0    2    0    0    0) /ExtU
O10  (  Kakucho, 50.0    2    0    0    0) /ExtU
O10  (  Kakucho, 58.0    2    0    0    0) /ExtU
 C3  (Shinryaku, 40.0    2    0    0    0) /Inv33
 I4  (Shinryaku, 43.7   74    0    0    0) /InvMiddle
 G4  (     Kiri, 25.0    6  200    0    0) /Cut1
 G6  (     Kiri, 25.0  200  136    0    0) /Cut1
 C4  (     Kiri, 25.0   30    6    0    0) /Cut1
 A6  (     Yose, 26.0    0    0    0    0) /HaneYose
B10  (     Yose, 45.0    0    0    0    0) /KosumiYose
N18  (     Yose, 45.0    0    0    0    0) /KosumiYose
R16  (     Yose, 45.0    0    0    0    0) /KosumiYose
B15  (     Yose, 45.0    0    0    0    0) /KosumiYose
 Q2  (     Yose, 36.0    0    0    0    0) /OsaeYose
  :
  :
```

Fig. 4 Examples of patterns in Gopal programs and their execution.

4. Concluding Remarks

In this paper we have described an approach to conquer the difficulties in developing a Go playing program. We have separated the difficulties on programming technique from ones the game of Go inherently has, and we have tried to reduce the difficulties in the first class.

In the first half of this paper, we discussed the formal specification of the interface between the data management module and other upper level modules. Using this interface, we have hidden the details in the data management module. This modularization technique is known as information hiding or data abstraction, and we have strictly applied this technique by using our Pascal system which supports such module structures. The description of the formal specification, however, is merely an informal document. Although there exist some modern languages such as Modula-2 and Ada which allow users to write both definition and implementation modules, the definition does not include full specification. A language in which we can describe the formal definition and the implementation, is definitely required for such an approach.

In the last half of this paper, we have discussed a non-procedural language for the Go pattern knowledge and its processor. The Gopal programs are also human-oriented, and it is easy to write and update a set of the pattern knowledge in this language. This is one of the examples which show that the non-procedural language designed for a specific application works very well. It would be better if we were able to write both efficient procedures and non-procedural statements in the same language.

Our Go program is now in development and we still have difficult problems to solve such as the estimation of the global state of things, and the creation of plans based on a strategy. These are, however, the problems Go inherently has, and we hope that we have already had a breakthrough. Then we will be able to attack and solve these problems using routines developed up to now by applying our approach, such as routines in the data management module, the Gopal library, expert routines and those to approximately estimate the board status.

The author wishes to thank the members of our Go project for many helpful discussions. The author also thanks Mr. Takamune who has assisted in implementing Gopal.

References

1. SANECHIKA, N. Recent Development in Game Playing Programs, *Information Processing* **20**, 7 (July 1979), 601–611 (in Japanese).
2. REITMAN, W. and WILCOX, B. The Structure and Performance of the INTERIM. 2 Go Program, *6th IJCAI* (1979), 711–719.
3. ZOBRIST, A. Feature Extraction and Representation for Pattern Recognition and the Game of Go, PhD. Thesis, University of Wisconsin, 1970.
4. SANECHIKA, N. et al. Notes on Modelling and Implementation of the Human Player's Decision Processes in the Game of Go, Bul. Electrotech. Lab., **45**, 1981.
5. SUGAWARA, Y. and SANECHIKA, N. A Method to Recognize the Strength Situation in Go Program, *23th National Conf. of IPSJ* (Oct. 1981) (in Japanese).
6. SANECHIKA, N. Programming the Decision Processes in Go, *Tech. Rep. of IPSJ* (Jun. 1982) (in Japanese).
7. BENSON, D. B. Life in the Game of Go, *Inf. Sci.* **10**, 1 (1976), 17–29.
8. BENSON, D. B. and SOULE, S. P. Legal Go: A Formal Program Specification, Part 1, 2, 3, Washington State Univ. CS-78-45, 46, 47 (1978).
9. DIJKSTRA, E. W. Notes on Structured Programming, in "Structured Programming" Academic Press, 1972.
10. LISKOV, B. Programming with Abstract Data Types, *SIGPLAN Notices* **9**, 4 (Apr. 1974), 50–59.
11. PARNAS, D. L. A Technique for Software Module Specification with Examples, *Comm. ACM* **15**, 5 (May 1972), 330–336.
12. JENSEN, K. and WIRTH, N. Pascal—User Manual and Report, *Lecture Notes in Comp. Sci.* **18**, Springer Verlag, 1974.
13. WIRTH, N. Modula-2, ETH, Institut fur Informatik, 1980.
14. MANO, Y. A Technique for Extending Pascal and an Application to Pascal with Module Structures, to appear in *Trans. of IPSJ* (in Japanese).

(Received May 2, 1983; revised March 5, 1984)