AN ADA LIBRARY FOR POSITIONAL BOARD GAMES

by

ATHANASSIOS ANASTASSIOS MANGOLAS

Thesis submitted to the Faculty of the

Virginia Polytechnic Institute and State University

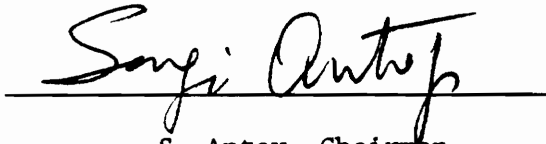
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

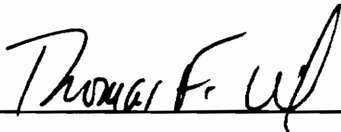
in

Computer Science

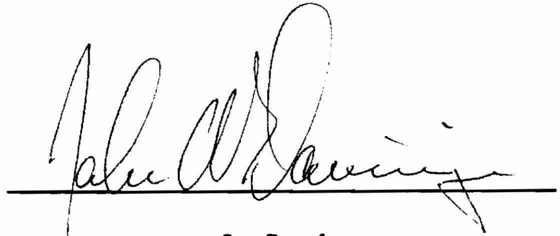
APPROVED:



S. Antoy, Chairman



T. Reid



J. Davison

July, 1990

Blacksburg, Virginia

c.2

LD
5655
V855
1990
M365
C.2

AN ADA LIBRARY FOR POSITIONAL BOARD GAMES

by

Athanassios A. Mangolas

Committee Chairman: Sergio Antoy
Computer Science

(ABSTRACT)

This thesis describes the design and the implementation of an Ada library for positional board games. The library consists of general software modules that use some concepts from a model of Positional Board Games presented in [Antoy 87]. This thesis shows that general software modules based on the mathematical concept of board can be built and used by any positional board game program.

Furthermore, this thesis describes the data types used in the modules and presents informal and formal specifications of the operations on the data types. It also describes the implementation of the data types; presents the algorithms implementing the operations and shows how the library can be used on two positional board programs and justifies the claim of generality and simplicity of the model in [Antoy 87].

The programming language Ada is used to express the formal specifications and to code the software modules.

ACKNOWLEDGEMENTS

After completing this work, I have a list of people to whom I owe genuine thanks. To my parents Maria and Anastassio Mangola, I own the most. They have served as ardent supporters throughout my studies. I have to thank my brother Nick for all the confidence and care, he has shown me. My advisor, Sergio Antoy, has my deep appreciation. Our day-to-day contacts have been delightful. He has been supportive, full of ideas and humor. His intellectual influence on me has been significant. Finally, I come to my friend Ted Belay. Ted has been a critic, a supporter, and a friend in need. I share with him enthusiasms for all kinds of things and ideas. For that I have to thank him deeply. I want to apologize for all those persons that are missing from this list and have helped me directly or indirectly during this work. The only reason for that is the profound contribution of the people mentioned.

TABLE OF CONTENTS

SECTION	TITLE	PAGE
ABSTRACT		ii
ACKNOWLEDGEMENTS		iii
LIST OF FIGURES		viii
Chapter 1.	**Introduction**	1
	1.1 Problem and Motivations	1
	1.2 Thesis Goals	3
	1.3 Thesis Organization	4
Chapter 2.	**Positional Board Games**	7
	2.1 Games in Artificial Intelligence	7
	2.1.1 Games in General	7
	2.1.2 Games as State-Space Problems	9
	2.1.2.1 Definition of State-Space Problems	9
	2.1.2.2 Game Trees	11
	2.1.3 Search in Problem Solving	12
	2.2 Positional Board Games	13
	2.2.1 Definition of a Positional Board Game	13
	2.2.2 A Model for Positional Board Games	13
	2.2.2.1 Term Definitions	14
	2.2.2.2 Model Characteristics	20

Chapter 3. **Design of Major Software Modules** 21

3.1 Design Principles 21

3.2 Library Description and Significance 23

3.2.1 Module Names and Module Interdependencies . 24

3.3 Modules Specification 27

3.3.1 Package Board 27

3.3.2 Package Configuration 32

3.3.3 Package Move 35

3.3.4 Package Game and Procedure X_game 37

Chapter 4. **The User Interface Package** 41

4.1 Scope and Design Goals of the Package

User_interface 41

4.2 Specification of the operations of the package

User_Interface 43

4.3 Implementation Issues of the Package

User_interface 47

4.3.1 Data Types, Data Objects, and Subprograms . 47

Chapter 5. **Theoretical Issues of the Implementation** 55

5.1 Move Selection

5.2 The Search Algorithm Alpha-Beta 55

5.3 Heuristics in Alpha-Beta 57

5.4 Major Concepts 59

5.5 Termination Criterion of the Alpha-Beta Algorithm 61

5.6 The static evaluation function **evaluate** 62

5.7 Reordering Criterion 68

Chapter 6. **Implementation of the Games of Hex and Go-Moku** 74

6.1 Game Specific Elements of a Positional Board Game Program 74

6.2 Game of Go-Moku: Game Definition and Win Group Computation Algorithm 77

6.2.1 Definition of the positional board game of Go-Moku 77

6.2.2 Win Group Computation Algorithm for the game of Go-Moku 78

6.3 Game of Hex: Game Definition and Win Group Computation Algorithm 80

6.3.1 Definition of the positional board game of Hex 80

6.3.2 Identification of the usable and useful win groups 82

6.3.3 Win Group Generation for the game of Hex .. 88

Chapter 7. **Programming Issues of the Implementation** 97

7.1 Implementation Principles 97

7.2 Implementation of the Package body Game 98

7.3 Implementation of the Package body Configuration 100

7.4 Implementation of the Package body Move 101

Chapter 8. **Conclusion** 113

8.1 Results 113

8.2 Conclusion 115

Chapter 9. **Literature Cited** 116

Appendices

Appendix A. Computer Program Listings 119

Appendix B. Package board Listings 161

Vita 184

LIST OF FIGURES

FIGURE	TITLE	PAGE
2.1	(a) Physical board and (b) Board characteristics of the game of Go-Moku	18
2.2	(a) Physical board and (b) Board characteristics of the game of Tic-Tac-Toe	19
3.1	Module Interdependencies	26
3.2	Specification of the package Board for the game of Tic-Tac-Toe	29
3.3	Specification of the package Configuration	34
3.4	Specification of the package Move	39
3.5	Specifications of (a) the package Game and (b) the procedure X_game	40
4.1	Specification of the package User_interface	52
4.2	Display of the game of Go-Moku (7x7)	53
4.3	Procedure init_display_coordinates for the game of Hex (4x4)	54
5.1	Alpha-Beta Game Tree Search Algorithm	71
5.2	Example of the static evaluation function evaluate	72
5.3	Example of relation LE	73
6.1	Procedure init_display_coordinates for the game of Hex (5x5)	91
6.2	Procedure init_display_coordinates for the game of Go-Moku (7x7)	92

6.3 Physical board for the 19x19 game of Go-Moku 93

6.4 Win Group Sets Generation Main Algorithm for the game of Go-Moku (NxN) 94

6.5 Orthogonal Array Representation of an 11x11 Hex physical board 95

6.6 Win Group Sets Generation Main Algorithm for the game of Hex (NxN) 96

7.1 Data structure pos_in_wga 111

7.2 Modified Alpha-Beta Game Tree Search Algorithm 112

CHAPTER 1

INTRODUCTION

Abstract

This chapter describes the subject and motivation of the thesis, the goals of the research, the work done and the thesis organization.

1.1 Problem and Motivations

The subject of this thesis is positional board games. It belongs to the field of Artificial Intelligence. The goal is to develop an Ada library of modules general enough to be used by any positional board game.

These software modules are general in the sense that they are common for the whole class of positional games and can used as elements of programs that play these games with as little modification as possible. Main characteristics of these modules are modularity, generality and reusability. Game specific information, in terms of move selection can be included as a separately compiled function.

Generality and reusability are achieved by parameterizing: (a) the number of board positions, and (b) the win-group sets. These game parameters mainly distinguish one game from another in terms of playing. In the design, this parameterization is expressed by the generic Ada package facility.

The main principles behind the design of the library are data abstraction and information hiding. The programming language Ada has been chosen for the implementation because it has the package data abstraction mechanism which is well suited for the above characteristics.

The major motivation for this work comes from [Antoy 87] where an abstract model of positional board games is presented. This model provides a framework that unifies the class of positional board games by introducing an abstract mathematical description of these games. With the help of the framework, the concept of isomorphism between two games is defined and properties about win-groups and isomorphic games are proven. The framework provides a general abstract definition of the class of positional board games by abstracting away visual characteristics of a game and the procedural rules that govern a play. These features are of great importance to this research.

[Antoy 87] is the foundation for the development of general modules. The key point is the concept of win group. In this work, there is a minor extension of this concept. We extend the model from one set of win groups common to both of the players to two sets of win groups. The win groups represent winning patterns of positions and can be used to define playing strategies. A strategy based on win groups is proposed and formalized.

Critical aspects of the move selection process like configuration-node reordering, dead-node elimination, and live-node identification are

proposed. They are all based on win groups. The major unifying idea for all of the above search-reduction criteria, as well as for the static evaluation function, is the distance from a win state.

1.2 Thesis Goals

The product of this work includes an Ada library of software modules that can be used to play positional board games.

The goal was to make this library general enough to be used by all positional board games. As such this work comprises the following parts:

1. The Design of the software modules.

The design consists of the selection of data types and operations common to all the games. It is expressed in terms of informal and formal specifications. Informal specifications are expressed using the English language. Formal specifications are expressed using the Package Specification Facility of the Ada programming language.

2. The Implementation of the Design.

The scope of the implementation is to show the validity of the design and the model presented in [2].

The implementation consists of: (a) the implementation of the data types and operations of the design, (b) the development of two positional board games to test the implementation and, (c) the development of a user-interface package for the library.

This thesis also demonstrates that a software system based on abstract mathematical structure of board in [Antoy 87] can be build with substantial gain in generality and simplicity.

1.3 Thesis Organization

This thesis is organized into the following chapters:

Chapter 2 describes the positional board games and defines some technical terms used in the thesis.

Chapter 3 presents the informal and formal specifications of the abstract data types and operations. Informal specifications are given in English while formal specifications are given in an Ada package form.

Chapter 4 proposes a User Interface common to all the games implemented and presents the specifications of the User Interface package which handles the I/O for all the games implemented. It also discusses the implementation of the basic operations provided by the package User

Interface.

Chapter 5 describes the mathematical basis of the static evaluation function and the heuristic rules used by the search algorithm **alpha-beta** to compute a move for a player.

Chapter 6 describes the programs used to create the win-group sets for the games of Hex and Go-Moku. The description is basically an exposition of the algorithms used to compute the win-groups. The mathematical basis of the algorithms is presented. An interesting characteristic of these programs is that their output is Ada code which is part of the library and constitutes the game specific information.

Chapter 7 presents the major data structures and subprograms inside the package bodies of the library and justifies a number of technical decisions.

Chapter 8 summarizes the product of this work and presents the conclusion.

Chapter 9 cites the literature used in this work.

Appendix A. contains the Ada source code of the whole library and the support programs.

Appendix B. contains the package **board** for the games of Hex and Go-Moku.

CHAPTER 2

POSITIONAL BOARD GAMES

Abstract

This chapter gives a brief introduction to games, provides a formal definition of positional board games, and defines major concepts used in the thesis.

2.1 Games and Artificial Intelligence

In this section we present general information about games.

2.1.1 Games in General

Games are heavily investigated topics in AI [Banerji 80, Feigenbaum 81, Jackson 74, Nilsson 80]. They provide a conceptually simple and stimulating vehicle where we can test ideas about reasoning, develop strategies and programs to demonstrate these ideas and test their validity in the most straightforward way: by measuring a program's playing performance against a human player. By observing the results, we can draw informal but fairly accurate conclusions about our ideas in machine intelligence.

Games are well defined problems. As such they possess well defined goals for the players and a set of rules that govern the permissible

actions. Each player may have his own set of goals and rules distinct from the opponent's. He tries to achieve the goals by applying rules to modify an existing situation. For each game there is an initial situation as a starting point. The selection of actions is the most challenging point from the AI point of view because there are no rules, in the definition of a game, to specify how to reach a goal. Since AI focuses on the identification and codification of the human problem solving ability, game playing fits the job. The results of this research are used for the development of "algorithms" which are used by programs playing these games. It is to be noted that a formal mathematical analysis of the "algorithms" is extremely hard if not impossible. Most of the programs, at least the most interesting ones, employ heuristic search to find a good solution rather than an algorithm employing an optimum strategy to find the best move. This is due to exponential nature of the problem.

Besides that factor, the identification of game characteristics shared by many games leads to the creation of game classes. Game classes when expressed in terms of a formal mathematical system help us prove useful properties about the games for game playing. This approach eases the task of game analysis by imposing a framework within which we look for game playing methods shared by all the games of a class. The attempt to find playing methods common to all the games in a class makes it an interesting task since it involves deep understanding of the common characteristics of the games in a class. These

characteristics help us create a model to describe the class and consequently to reason about it. The benefit from this approach is generality in the game playing methods employed, since only the common features, as presented in the formal model, are used. What is not guaranteed is the efficiency of these methods. This is true mostly because they omit non-common characteristics, like board symmetry in positional board games, that can be used to reduce the search space, and time.

2.1.2 Games as State-Space Problems

Games can be treated as special cases of state space problems. A brief introduction describing the major elements of such a treatment follows.

2.1.2.1 Definition of State Space Problems

A problem represented in a state-space formalism includes two sets as elements: (a) the set S of all permissible states (**state space**) and (b) the set O of the all those operators that transform one state to another in S. An operator is usually a partial function since its domain is a subset of S. A state is a composite object of all those characteristics needed to define a distinct situation in the problem. A game state will subsequently be called **configuration**. An operator is a

rule or a procedure that changes one configuration to another by modifying some of its characteristics in some way. An operator application (operation) to a configuration will subsequently be called **move selection**. The selection of both the states-their representation- and the operators is problem specific.

In the above formulation, some of the permissible states are defined as initial states while some others are defined as final states. The former constitute the starting point while the latter constitute the problem solutions. It is important to note that operators perform the transition from one legal state to the other and only that. We call **I** the set of all initial states and **F** the set of all final states. An initial state will subsequently be called **initial configuration** and a final state **terminal configuration**.

A solving process is one that allows the selection of a sequence of operators to a final state. So we seek a path from a given initial state to a final one. The selection of this path is among the major problems that AI examines. The exploration of the state-space can be achieved in many ways. Herein, we will concentrate on the search method for the identification of the path from an initial configuration to a final one. The shortest path to a final state is of fundamental importance in terms of efficiency.

State-space problems can be graphically depicted as directed graphs. In these graphs nodes correspond to states and arcs to operations that transform one state to another. The state at the tail of the arc is the parent state while the state at the head of the arc is derived from the parent state through the application of the operator labeling the body of the arc. In such a notation there is a state called starting state serving as the root of the state space tree. To create the state space of the problem, we apply all the permissible operators to the starting state in order to create its successor states. For each of the successor states, we do the same thing until we reach a final state where there are no operators applicable to that. The process is completed when all the leaves of the created tree are final states. This graphical notation is used to depict portions of the state-space in the following chapters.

2.1.2.2 Games Trees

This thesis deals with two-player games as mentioned in the preceding chapter. These games can be represented as state space problems. A state is a board configuration. There is always one and only one initial configuration which is the root of the space tree called **game tree** in this case. There is no element of chance involved in the transition from one configuration to the other. So an operator corresponds to a move selection. Successor states result from a move and

final states correspond to either a win for some player or a tie. A **complete game tree** represents all the possible distinct plays in a game [Feigenbaum 81]. Such a tree is generated as follows: (a) the root of the tree is the initial game configuration where a player, say A, has to select a move, (b) we apply all the permissible operators, in our case all the legal moves for player A, to this configuration getting the successor configurations where it is players' B turn to move, and (c) we do the same for each one of those successor configurations until we reach the point where all the leaf configurations in the tree are final configurations (win or draw).

2.1.3 Search in Problem Solving

Search is one way to view problem solving especially in a state space problem formulation. A search method creates a game tree until it finds a terminal configuration. In the problem (game) definition there is a starting state, a set of operators that create all the successor states of a state and a set of final states, beginning with the starting state we create the state space by applying a sequence of permissible operators to all the states until we have reached a final state. Basically we move in the state space looking for some final state. A search method that uses domain information to narrow down the size of the state space visited for a move is called **heuristic search**.

2.2 Positional Board Games

In this section, a detailed description of positional board games is given.

2.2.1 Definition of a Positional Board Game

A positional board game is a zero-sum, two-player, perfect-information, non-random game. It is played on a set called **board** whose elements are called **positions**. The players select alternatively positions to occupy on the board permanently. There is a set of rules that defines what a legal move is and when a player can claim win. Usually, a player can claim win when he has filled certain patterns of positions. A game starts with one's player move and terminates when either one of the players has won or there are no available positions to select.

2.2.2 A Model for positional board games

Most of the material presented in this section comes from [Antoy 87]. Major concepts and term definitions used throughout this work are provided.

2.2.2.1 Term Definitions

A **Board B** is comprised of a set **P** of positions and two sets **WGA** and **WGB** of win groups, one for each player. The set of positions has cardinality N. Positions are identified as 1, 2, ..., N. A win group w is subset of P. It defines a pattern of positions such that if all the position of w are occupied then the player to whom this win group belongs has won (**win**). The set of win groups for player A is denoted as WGA and for player B as WGB. Each win group can be qualified into four classes - sets - according to their importance for game-playing:

1. The set of **trivial win groups** is the set of all win groups of cardinality 1. Games that contain trivial win groups are called trivial games and they are of no interest.

2. The set of **useless win groups** is the set of win groups that do not need to appear in a terminal configuration to claim win for some player.

3. The set of **usable win groups** is the set of the win groups that if completely occupied by a player can be used to claim win.

4. The set of **useful win groups** is the set of all the win groups that

must appear in a terminal configuration, completely occupied by a player, to claim win.

If we have a board B =(P, WGA, WGB) and a subset f of P we define f as a free subset iff no win group is contained in f.

So a board B is denoted by (P, WGA, WGB). A position can have three values: **E** which denotes that the position is empty, **A** which denotes that the position is occupied by player A, **B** which denotes that the position is occupied by player B. A **configuration C** is an assignment of a value s in {E, A, B} to each position p in P. If for each p in P, p is assigned the value E then this configuration is called **initial configuration**. If in a configuration C_t each position p in P is assigned a value s in {A, B} or there is a win then this configuration is called **terminal configuration**. A sequence of moves <m> generates a sequence of configurations <C> as follows:

If <m> = $\{m_1, m_2, \ldots, m_t\}$ is a sequence of moves and

C_0 is the initial configuration

then the configuration sequence

<C> = $C_0, C_1, \ldots, C_t$ is defined as follows:

for k in the range $1 \le k \le t$ and $1 \le p \le N$

$$C_k(p) = \begin{cases} C_{k-1}(p) & \text{if } p \neq m_k, \\ A & \text{if } p = m_k \text{ and } k \text{ is odd}, \\ B & \text{if } p = m_k \text{ and } k \text{ is even}. \end{cases}$$

The sequence <C> is called a **game** if C_t is the only terminal configuration in <C>. A game <C> is the shortest for a win group w if and only if the positions occupied by player A in C_t are those in the win group w which belongs to the win group set WGA.

If X is in {A, B} and w is a win group for X and for every p in w p is in {X, E} then w is called **open win group**.

A position p in the win group w is called **hole** if w is an **open win group** and p is in {E}.

The following propositions concerning the win groups board positional games were proved in [Antoy 87]:

Proposition 1

There exists a shortest game for a win group w iff w is usable.

Proposition 2

A usable win group w is useful iff no win group is properly contained in w.

Proposition 3

A win group w is usable iff there exists a free subset of P - w and a free subset of w, both of cardinality $|w| - 1$.

What the above propositions suggest is that for game-playing purposes we need to consider only the useful win groups and ignore the others. Furthermore we may not include usable and useless win groups at all in a game playing program since there need not be used to claim a win because a useful win group can be used. Figures 2.1 and 2.2 provide definitions of the positional board games Hex/3 and Tic-Tac-Toe according to the above formalism.

```
(a)   1 - 2 - 3
      | / | / |
      4 - 5 - 6
      | / | / |
      7 - 8 - 9
```

(b) P = {1, 2, 3, 4, 5, 6, 7, 8, 9}

WGA={ {1, 4, 5, 6, 9}, {1, 4, 5, 8}, {1, 4, 7},
{2, 4, 7}, {2, 5, 6, 9}, {2, 5, 7},
{2, 5, 8}, {3, 5, 7}, {3, 5, 8}, {3, 6, 8}, {3, 6, 9} }

WGB = { {1, 2, 3}, {1, 2, 5, 6}, {1, 2, 5, 8, 9},
{4, 5, 6}, {4, 5, 3}, {4, 5, 8, 9},
{4, 2, 3}, {7, 8, 9}, {7, 8, 6}, {7, 5, 6}, {7, 5, 3} }

Figure 2.1

(a) Physical board and (b) Board characteristics of the game of Hex.

(a)

```
1 | 2 | 3
---------
4 | 5 | 6
---------
7 | 8 | 9
```

(b) P = {1, 2, 3, 4, 5, 6, 7, 8, 9}

WGA = WGB = { {1, 2, 3},
{4, 5, 6}
{7, 8, 9},
{1, 4, 7},
{2, 5, 8},
{3, 6, 9},
{3, 5, 7},
{1, 5, 9} }

Figure 2.2

(a) Physical board and (b) Board characteristics for the game of Tic-Tac-Toe.

2.2.2.2 Model characteristics

This model reduces the differences between two positional board games into three sets, namely P, WGA, WGB. Furthermore, it provides a uniform way to define all the games in the class. The cardinality N of set P and the sets WGA and WGB can be considered as parameters by program modules that implement basic operations common to all the positional board games leading the way to general software modules that do not depend on the procedural definition of each separate game.

Board configurations can be represented as a one dimension array of size N, where each element assumes values in {E, A, B}.

The win group sets WGA and WGB can also be represented as one dimension arrays whose sizes are the respective cardinalities of these sets of win groups. Each element of the above arrays is a win group.

CHAPTER 3

DESIGN OF MAJOR SOFTWARE MODULES

Abstract

This chapter presents the design of the software modules of the Ada library for positional board games. We list the software modules of the library, discuss the main concept and the scope behind each module and describe the data types, data objects and operations that each module provides. To accomplish this description we use informal English and formal Ada package specifications. We also discuss design principles.

3.1 Design Principles

The major objectives of this work are reflected in the design of the modules. The design captures the definition and the basic game playing operations of any positional board game in a way that achieves generality and reusability.

We need to define the goals mentioned in Chapter 1 more precisely in order to show that the design reflects those goals. **Generality** is the property shared by the modules that permits them to be parts of programs that implement different positional board games as they are, without modifications. In the present context , the concept of **reusability** is essentially the same as the concept of generality since the same software modules can be used by different programs without modification. From a

theoretical point of view, generality refers mostly to the underlying mathematical foundation of the modules while reusability refers to the actual modules and it is achieved because of the generality. Finally, **modularity** refers to the overall organization of the library into independent components. The number of major modules has been kept small enough to achieve simplicity. It is to be noted that this goal is not in full accordance with generality. Generality has been given a higher priority than modularity.

Abstraction is the modelling principle that captures only the essential, leaving the details out, features of a concept that are used for its definition and distinction from other concepts. In this way we can group related objects into classes. **Data abstraction** is the application of the above principle to data types. Specifically, data abstraction captures the representation and operations of a data type. For the sake of completeness we mention that a **data type** is a set of objects and operations on those objects. The **Specification** of a module describes the data types and operations that the module makes available to other modules. The implementation of all the operations is kept hidden. Where possible, the implementation of data types is also kept hidden. This informal description is given in English and is complemented by a formal specification in an Ada package form.

Now we define the concepts of the Ada language that are used in this

work. **Package** is a program unit that contains the specification of groups of related entities. These entities can be data types, data objects (constants, variables), subprograms, other packages and tasks. In this context when we refer to a module we mean either a package or a subprogram.

3.2 Library Description and Significance

The Ada library that has been developed for the positional board games captures the common representational and operational aspects of the class of the positional board games. This work proved the claim that the model of [Antoy 87] can be the foundation for building general software modules common to all positional board games. The design of the modules constitutes the proof of the above claim. Furthermore the small size of the library and its clear and concise specifications show that not only there is a gain in generality and reusability by using the model of [Antoy 87] but also in simplicity and size. The two games which were implemented using the library demonstrate this claim by showing that the difference between the two programs that implement these games lays on the package **board** which contains that definition of each game.

The design of each package is based on a single concept. The contents of the package support the various aspects of the concept. The package specification makes these aspects visible. The identification and

separation of the main concepts related to game playing for positional board games is the major part of the design and it is described together with the module specifications.

3.2.1 Module Names and Module Interdependencies

The library consists of six Ada modules: (a) the package **board**, (b) the package **configuration**, (c) the package **move**, (d) the package **game**, (e) the package **user_interface**, and (f) the procedure **x_game**. The package user_interface is described in detail in a separate chapter. The remaining five modules are described in the current chapter. The interdependencies between the modules are shown at the Figure 3.1. The modules are represented by their names. A directed arc is used to show the dependency between two modules. The depended module is at the tail of the arc while the independent module is at the head. We explain these dependencies together with each module's informal specifications.

Note that the interdependencies are between Ada program units and as such they cannot be cyclic, for example if the specification of program module A depends on the specification of program module B then the specification of program module B cannot depend on the specification of program module A too. Now if some data-types, data-objects or subprograms inside the body of program module B that do not appear in the specification part of B depend on the specification part of program module

A then this dependency cannot appear in the specification part of B but in the implementation part of it- package body of B. To conclude, not all the interdependencies are visible in the specification of the library. We will describe the interdependencies in more detail in Chapter 5 where we discuss the implementation.

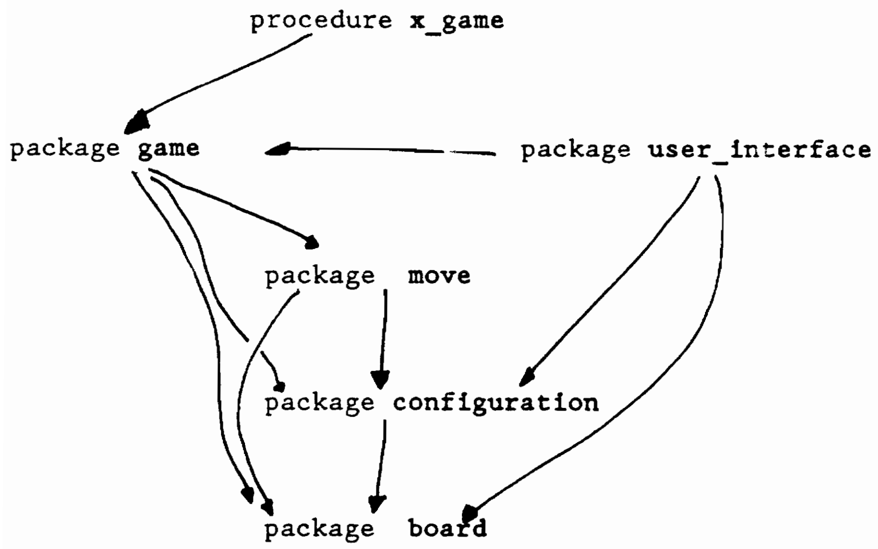


Figure 3.1

Module Interdependencies

3.3 Modules Specification

In this section we present the informal specifications, given in English, for each one of the modules. The description of a module is comprised of: (a) the purpose of the module, (b) the dependencies on other modules, (c) the description of data types and variables, and (d) the description of subprograms. The formal specifications for each module are also given in an Ada package form. The formal specifications appear in figures.

3.3.1 Package Board

Purpose

This package contains the definition of a positional board game. The package **board** represents exactly the mathematical concept of *board*, given in [Antoy 87] and presented in Chapter 2. This concept is described in this package by the constants *wga* which denotes the set of win groups WGA, *wgb* which denotes the set of win groups WGB and the constant *number_of_positions* which denotes the cardinality $|P|$ of the set P of the positions. Each program that implements a positional board game has to have a corresponding package **board** that contains the game's definition in terms of the model. Two distinct positional board games vary on the corresponding packages **board**. The difference between the two packages is

at the values of the variables *number_of_positions*, *wga*, and *wgb*.

The package **board** also includes the data types of the above data objects that are needed to store the description of a positional board game. The package **board** does not depend on any other module in the library.

Figure 3.2 presents the definition of the game Tic-Tac-Toe in an Ada package specification form.

```
package board is

type wg_array is array(integer range <>) of integer ;

type wg_type(size : integer := 3) is record

     wg : wg_array(1.. size) ;

     end record ;

type wg_table is array(integer range <>) of wg_type ;

number_of_positions : constant natural := 9 ;

type values is (player_a, player_b, empty, not_available ) ;

subtype board_range is integer range 1..number_of_positions ;


wga : constant wg_table(1..8) := (

(3,(1,2,3)), (3,(4,5,6)), (3,(7,8,9)),

(3,(1,4,7)), (3,(2,5,8)), (3,(3,6,9)),

(3,(1,5,9)), (3,(3,5,7)) );


wgb : constant wg_table(1..8) := (

(3,(1,2,3)), (3,(4,5,6)), (3,(7,8,9)),

(3,(1,4,7)), (3,(2,5,8)), (3,(3,6,9)),

(3,(1,5,9)), (3,(3,5,7)) ) ;

end board ;
```

Figure 3.2

Specification of the Package Board for the game of Tic-Tac-Toe

Data Types, Constants and Variables

Data Type wg_array

The data type **wg_array** is a component of the **wg_type** defined below. It is used to declare data objects that store the positions of a win group. This type is defined as a one dimension unconstrained array of integers which represent positions. The index of the array ranges from 1 to <u>size</u> where <u>size</u> is provided for each data object separately. This type of array is called **dynamic array**.

Data Type wg_type

The data type **wg_type** is used to declare data objects that store a win group. It is declared as a two-field record where: (a) *size* of data type **natural** is a discriminant and denotes the size of the win group, and (b) *wg* of data type **wg_array** is the field where we store the positions of the win group. The *size* field determines the size of the array *wg*.

Data Type wg_table

The data type **wg_table** is used to declare data objects that store the set of win groups. It is declared as an unconstrained array whose elements are of wg_type.

Constant number_of_positions

The constant **number_of_positions** is declared as natural and denotes the size of the board which is the cardinality of the set P of positions.

Subtype board_range

This data type is defined as the integers in the range 1 to number_of_positions. It is used to define data objects that take values in P.

Data Type values

This is an enumerated data type which denotes the permissible values of a position (position states). The values are **player_a** which denotes the value of a position occupied by player A, **player_b** which denotes the value of a position occupied by player B, **empty** which denoted the value of an empty position, and **not_available** which denotes that value of a position that is not occupied and cannot be occupied by any of the players currently. This provision constitutes an extension of the model in [Antoy 87].

Constants wga and wgb

The constants **wga** and **wgb** of data type is **wg_table** denote the sets of win groups for players A and B respectively. During declaration, the two constants are initialized with the corresponding sets of win groups for the two players.

3.3.2 Package Configuration

Purpose

The package **configuration** stores and maintains the board configuration; this is its sole purpose. It provides access to the board configuration only through four operations. The representation of the board configuration is kept hidden.

This package depends on the package **board** for the: (a) declaration of the data type **values** of the values of a position, (b) declaration of the variable **number_of_positions** which denotes the size of the board, and (c) declaration of the subtype **board_range**.

Subprograms

procedure clear_board

This procedure generates the **initial configuration**. It does so by assigning the value **empty** to each board position.

procedure put_value(v,p)

The procedure put_value assigns the value v to the position p. A CONSTRAINT_ERROR exception is raised when p is not assigned a value in board_range or when v is not assigned a value of the data type **values**.

procedure unput_value(p)

The procedure unput_value assigns the value **empty** to position p. If p is not assigned a value in board_range then the exception CONSTRAINT_ERROR is raised.

function get_value(p) returns v

The function get_value returns the value v that is assigned to position p. If p is not assigned a value in board_range then the exception CONSTRAINT_ERROR is raised.

Figure 3.3 presents the Ada package specifications for the module **configuration**.

```
package configuration IS

  procedure clear_board;

  procedure put_value(element : IN values; position : IN board_range);

  procedure unput_value(position : IN board_range) ;

  function  get_value(position : IN board_range) return values;

end configuration;
```

Figure [illegible].3

Specification for the Package Configuration

3.3.3 Package Move

Purpose

The package **move** handles the move identification (search for a move) and move selection aspects of game playing for position board games. It provides two operations: an operation called **find_a_move** which finds a move to be chosen for any of the two players in a game without modifying any game-playing characteristics, like configuration, open win groups, and players' turn to move, and an operation called **make_a_move** that makes a move to a position and updates game-playing characteristics. This package depends on the packages **board** and **configuration**. The operations implemented in the package **move** access the board configuration through the operations of the package **configuration**. Similarly they depends on the game definition, like the sets of the win groups and number of board positions, found at the package **board**.

Exceptions

There are three exceptions declared in the package **move**: **winner_a**, **winner_b**, and **draw**. These correspond to the terminal configuration conditions: win for player A, win for player B, and draw.

Procedures

find_a_move(m)

This procedure searches for a "good" position **p** to move for the

player whose turn is to move in a game. It assigns the suggested position p to m. If this procedure is called when the current configuration is terminal it raises the appropriate exception that indicates what kind of terminal condition -win or draw - has occurred.

make_a_move(1,p)

This procedure selects the move to the position **p** by player **1**. It does so by assigning the value 1 to the position **p** and updating the open win group information stored inside the body of the package **move**. If the configuration that results from the move to **p** is terminal then the corresponding exception is raised, **winner_a** (**winner_b**), if player A (player B) has won or **draw**. When one of the above three exceptions is raised a corresponding exception handler is invoked which prints an appropriate message about the event that caused the exception, clears the board configuration by assigning it the **initial configuration** value, and initializes data objects that hold game-playing information like the number of moves made, the first player variable, and the status of the win groups. The status of the win groups refers to which win groups are **open** and how many **holes** exists in each of these open win groups.

Figure 3.4 displays the formal specifications for the package **move**

3.3.4 Package Game and Procedure X_game

Purpose

The role of the package **game** is to let the user of the library play the positional board game defined in the package game. This is done through the operation **play_game** which connects all the other packages. It uses the package **board** for the game definition, the move operations of the package **move** , the package **configuration** to clear the configuration and the package **user_interface** to perform I/O functions. We discuss how this operation brings the rest of modules together in Chapter 5.

Data Types

The enumerated data type **options** is defined in order for the package **game** to communicate with the package **user_interface** the user selections from the menu that displays the game playing features. The values of the data type **options** represent user-selectable game-playing features which are independent from the display input/output operations and their implementation. We have assigned the following meaning to the values of this data type:

nothing : an invalid input was entered by the user

new_game : start a new game and initialize all the relevant program structures.

change_player : switch the user from player A to player B or vice versa.

play : accept a valid user move to a position and compute a move for the

computer and make the two moves.

suggest : find and report a suggested move for the user.

quit : terminate the program.

Procedures

play_game

This procedure plays from start to the end the positional board game defined at the package **board**. To execute this procedure we have defined the program unit **procedure x_game** which simply calls the procedure **play_game**. This procedure is not an independent program unit, as the procedure **x_game**, because we need to define the data type **options** and make it visible to the package **User_interface**.

Figure 3.5(a) shows the formal specifications of the package **game** and figure 3.5(b) shows the formal specifications of the procedure **x_game**.

```
with configuratio ;

use configuration ;

with board ;

use board ;


PACKAGE move  IS

winner_a, winner_b, draw : exception ;


  PROCEDURE find_a_move(suggested_move : OUT board_range) ;

  PROCEDURE make_move(player : in values ; move : in board_range) ;


END move ;
```

Figure 3.4

Specification of the Package Move

(a)

```
WITH configuration ;
USE configuration ;
WITH board ;
USE  board ;
WITH move ;
USE move ;

PACKAGE game  IS

    TYPE options IS (quit,play,nothing,suggest,new_game,change_player) ;
     PROCEDURE play_game ;

END game ;
```

(b)

```
with game ;
use game ;
  PROCEDURE x_game IS
      play_game ;
  END x_game ;
```

Figure 3.5

Specifications of (a) the Package Game and (b) Procedure X_game

CHAPTER 4

THE USER INTERFACE PACKAGE

Abstract

This chapter documents the design and the implementation of the package **User_Interface**. The package **User_Interface** handles all the Input/Output operations from and to the screen. We discuss the design goals, present the specifications of the operations provided by the package, and describe the basis of the implementation of these operations.

4.1 Scope and Design Goals of the Package User_interface

The package **User_interface** has been developed in order to provide basic screen Input/Output operations for the programs that implement the positional board games of Hex and Go-Moku. As we have mentioned in previous chapters these two game programs are based solely on the library modules described in Chapter 3. The implementation details of these two programs are discussed in Chapter 6. In this chapter we present the approach we followed towards the development of the **User_Interface** package. This package can be also used by any other positional board game program based on our library if its physical board can be represented as a two dimensional array. This restriction has been imposed by the character-based system I/O library **curses** which has been used to implement the operations supplied by this package and the physical characteristics

of the display available.

The main idea behind the package **User_interface** is encapsulate all I/O operations needed by the games implemented in a package, providing a common calling interface to these operations which does not depend on any specific I/O library. This is necessary in order to preserve the library properties of generality, modularity and reusability. The presence of specific I/O statements inside any of the library packages would have hampered the modularity of the library by making the library directly depended on a variety of I/O packages. An additional benefit from this approach is that we can easily upgrade the character-based display to a graphics-based display, provided an appropriate graphics library, by just changing the implementation of the operation inside the package body of **User_Interface**, leaving the other library packages intact. This kind of upgrade is both visually appealing and useful since not all the physical boards of the positional board games can be displayed on a character-based screen, though its development was outside the scope of the present work.

Section 4.2 presents the specifications of the screen I/O operations of the library and section 4.3 addresses the implementation issues. Also in section 4.3 we propose a method that confines the display differences among the physical boards of various games to the values of three arrays.

4.2 Specification of the operations of the package User_interface

Dependencies

The package **User_interface** depends on: (a) the library package **configuration**, (b) the library package **game**, (c) the library package **board**, (d) the system package **curses**, and (e), the system package **integer_io**. It uses the operation **get_value(p)** of the package **configuration** to get the value of a position p in a configuration. It uses the data type **options** of the package **game**, the data type **values** and the data object **number_of_positions** of the package **board**, the operation **put(int)** of the system package **integer_io** to write the integer **int** on the display and finally it uses the system package **curses** for all the other primitive input/output operations that manipulate windows on the screen. Figure 4.1 shows the specifications of the operations supplied by package **User_interface**.

Curses and Windows

Before proceeding with the description of the specifications we need to say that **curses** is a Unix Screen I/O library that allows direct screen manipulation. It allows the creation of multiple overlapping character-based windows on the screen and cursor control. The windows created by curses can be either visible or invisible and can be moved all over the screen. We have used some subprograms from the package **curses** to: (a) create and manipulate windows on the screen and (b) place the cursor to a

specific location inside any of the windows defined. We have defined the following three windows:

board_win : number of lines = 18, number of columns = 45,

options_win : number of lines = 18, number of columns = 22,

status_win : number of lines = 4, number of columns = 72.

Figure 4.2 show the display of the game Go-Moku/7.

We propose a display model that can be used to define the display of a physical board of a positional board game. The model is expressed in terms of the following data objects: **b_ch**, **total_lines**, **total_columns**, **xy_display**, **background**, and **xy_background**. We subsequently describe the scope of each one of these.

Subprograms

Procedure init_display

The procedure **init_display** initializes curses library parameters related to the physical screen, clears the physical screen and creates three logical windows: *board_win* window which is used to display the physical board of a positional board game, *status_win* window which is used to display various messages and prompts to the user, and *options_win* window which is used to display the game-playing features that control the play of a positional board game and can be modified by the user. These

features are represented as values of the data type **options** mentioned earlier in Chapter 3.

Procedure clear_display

The procedure **clear_display** clears the physical screen.

Procedure display_board

The procedure **display_board** displays the physical board of the positional board game implemented on the window *board_win*.

Procedure display_menu

The procedure **display_menu** displays the menu of the game features on the window *options_win*.

Procedure wait_for_a_keystroke

The procedure **wait_for_a_keystroke** displays the message: "Hit any key to proceed !" and suspends the program executions until the user hits a key.

Procedure print_winner(w)

The procedure **print_winner** has one IN parameter **w** of the data type **values** and writes a message on the window *status_win* depending on the value of **w**. The parameter **w** denotes either a win or a draw. The values **player_a**, **player_b** denote the winner of the game and the value **empty**

denotes a draw.

Procedure update_tip_nodes(t)

The procedure **update_tip_nodes** writes the message:"Tip nodes evaluated =>" and the value of the parameter **t** of type **natural** on the window *status_win*.

Function get_user_selection

The function **get_user_selection** displays the message: "Select ?" moves the cursor after that message and waits for the user to type in a character in the window **options_win**. The character typed in must be among those characters that are enclosed in parentheses in the window **options_win**. The user does not need to hit the key **ENTER** after typing in a character. There also is no distinction between uppercase and lowercase characters. Each of these characters is mapped into a value of the data type **options**. It is this value of the data type **options** that is returned to the caller of this function.

Function get_player_move

The function **get_player_move** displays the messages: "Enter Move!", "Line =>", and "Column =>", and gets the *line* and a *column* coordinates of a board position displayed on the window **board_win**. The user must hit the **SPACE BAR** key after typing in the number that corresponds to either a line or a column of the board displayed. A *line* or *column* out of bounds

is not accepted. A valid combination of *line* and *column* coordinates is mapped to a **board** position **p**. The position **p** is return to the caller of the function.

4.3 Implementation Issues of the Package User_interface

In this section we discuss the implementation of the procedures **display_board**, **display_background** and **init_display_coordinates** and the major data types and data objects of the package that are used by these procedures.

4.3.1 Data Types, Data Objects, and Subprograms

General

The current implementation of the package **User_interface** can only display the physical boards of those positional board games that can be played on a two-dimensional **LxC** board that resides on a plane. Such a board is divided into **LxC** cells where each cell represents a position. To each one of the cells we assign a pair of screen coordinates **(lp,cp)**, where **lp** denotes the line and **cp** denotes the column, in the window **board_win**. This pair of coordinates specifies a location in the window **board_win** where a character denoting the value of the position represented by the cell is displayed.

We associate an array of characters called **background** of size **b_ch** to each physical board of a positional board game. These characters are called **background characters**. They are written between physically adjacent board positions to show that two positions are adjacent. The characters in the array **background** takes values from the set {'/', '\', '|', '-', '*'}. We associate a pair of screen coordinates (bl, bc), where bl denotes a line and bc denotes a column, in the window **board_win** to each one of the components of the array **background** that specifies the location where this component is written.

Data types and data objects

Data type xy_cord

The data type **xy_cord** is a record data type with two fields: **x** and **y** both of them of the data type **natural**. A data object of this data type stores a pair of actual screen coordinates in the window *board_win*. The field **x** denotes a line and the field **y** denotes a column.

Data type xy_table

The data type **xy_table** is a one dimensional unconstrained array where each component is of the data type **xy_cord**. It is used to declare the data object **xy_display**.

Data object xy_display

The data object **xy_display** is of the data type **xy_table**. The size

of the array is $|P|$ = **number_of_positions**. The component **xy_display(p)** of the array correspond to the board position **p** in **P** and stores the screen coordinates in the window *board_win* where a character representing the value of the position **p** in a configuration **C** will be displayed.

Data objects background and b_ch

The data object **background** is an array of 400 characters. Each component of this array stores a **background character**. The data object **b_ch** is of the data type **natural** in the range 0 to 400 and denotes the number of the background characters stored in the array **background** that are written on the window *board_win*. The first **b_ch** components of the array **background** are used. The value of the data object **b_ch** depends on the physical board of each positional board game.

Data object xy_background

The data object **xy_background** is of the data type **xy_table**. The size of the array is 400. The component **xy_background(i)** of the array **xy_background** is associated with the component **background(i)** of the array **background** for every i in the range 1 to **b_ch**. The character stored in background(i) is written at the screen location of the window *board_win* whose coordinates are stored in **xy_background(i)**.

Data objects total_lines and total_columns

The data objects **total_lines** and **total_columns** are assigned values

of the data type **natural**. The data object **total_lines** keeps the number of lines displayed on window *board_win* and the data object **total_columns** keeps the number of columns displayed on window *board_win*.

Subprograms

Procedures display_board and display_background

The procedure **display_board** writes the value of each board position p in P at the location of window **board_win** specified by the coordinates: line = **xy_display(p).x**, column = **xy_display(p).y** and calls the procedure display_background. The procedure **display_background** writes the value **background(i)** for each i in the range 1 to **b_ch** to the location of the window **board_win** specified by the coordinates: line = **xy_background(i).x**, column = **xy_background(i).y**. The procedures **display_board** and **display_background** call **curses** subprograms to write on the screen. The code of these two procedures is given in the Appendix

Procedure init_display_coordinates

The procedure **init_display_coordinates** initializes the data objects **b_ch**, **total_lines**, **total_columns**, **xy_display**, **background**, and **xy_background**. The values given to these data objects depend of the characteristics of the physical board of the positional board game implemented. For each game implemented whose physical board can be represented in terms of the display model proposed the user of the library has to supply the appropriate procedure **init_display_coordinates**. This

procedure is declared as separate inside the package **user_interface**. Figure 4.3 displays the procedure **init_display_coordinates** for the game of Hex with 4 positions at each side.

```
with configuration; use configuration;
with game; use game;
with board; use board;
with curses; use curses;
with integer_io; use integer_io ;
package  use_interface is
   procedure init_display ;
   procedure clear_board ;
   procedure display_board ;
   procedure display_menu ;
   procedure wait_for_keystroke ;
   procedure print_winner(winner : in values) ;
   procedure update_tip_nodes(tips : natural) ;
   function get_player_move return natural ;
   function get_user_selection return options ;
end user_interface ;
```

Figure 4.1

Specification of the package User_interface

```
          window board_win                          window options_win
______________________________________________   _______________________
**********************************************   ***********************
*                                            *   *(N)ew Game           *
*                                            *   *(S)uggest Move       *
*   1 2 3 4 5 6 7                            *   *(P)lay               *
* 1 X X X X X X X                            *   *(C)hange Player      *
* 2 X X X X X X X                            *   *(Q)uit               *
* 3 X X X X X X X                            *   *                     *
* 4 X X X X X X X                            *   *                     *
* 5 X X X X X X X                            *   *                     *
* 6 X X X X X X X                            *   *                     *
* 7 X X X X X X X                            *   *                     *
*                                            *   *                     *
*                                            *   *                     *
*                                            *   *                     *
*                                            *   *                     *
*                                            *   *                     *
*                                            *   * Select ?            *
**********************************************   ***********************

                 area of window status_win
_________________________________________________________________________
```

Figure 4.2

Display of the game of Go-Moku

```
separate(user_interface)
procedure init_display_coordinates is
begin
  b_ch := 33;
  total_lines := 4;
  total_columns := 4;
  enum  := 1;
  FOR lines IN 1 .. total_lines LOOP
    FOR columns IN 1 .. total_columns LOOP
      pos_dir(lines, columns) :=  enum;
       enum :=  enum + 1;
    END LOOP;
  END LOOP;

 xy_display :=((4, 6), (4, 8), (4, 10), (4, 12), (6, 6), (6, 8), (6, 10),
(6, 12), (8, 6), (8, 8), (8, 10), (8, 12), (10, 6), (10, 8), (10, 10),
(10, 12));

xy_background(1 .. 33) := ( (4, 7), (4, 9), (4, 11), (5, 6), (5, 7), (5,
8), (5, 9), (5, 10), (5, 11), (5, 12), (6, 7), (6, 9), (6, 11), (7, 6),
(7, 7), (7, 8), (7, 9), (7, 10), (7, 11), (7, 12), (8, 7), (8, 9), (8,
11), (9, 6), (9, 7), (9, 8), (9, 9), (9, 10), (9, 11), (9, 12), (10, 7),
(10, 9), (10, 11));

background(1 .. 33) := ('-', '-', '-', '|', '/', '|', '/', '|', '/', '|',
'-', '-', '-', '|', '/', '|', '/', '|', '/', '|', '-', '-', '-', '|', '/',
'|', '/', '|', '/', '|', '-', '-', '-');
end init_display_coordinates ;
```

Figure 4.3

Procedure init_display_coordinates for the game of Hex (4x4)

CHAPTER 5

THEORETICAL ISSUES OF THE IMPLEMENTATION

Abstract

We describe: (a) the search algorithm *alpha-beta* that is used by the procedure **find_a_move** to compute a move for any of the two players, (b) some theoretical aspects of this algorithm that are game-depended, like *plausibility ordering, forward pruning*, *node liveness*, *static evaluation function*, and *dead-position identification*, and (c) the *heuristics* proposed to tailor the general *alpha-beta* search algorithm to our domain, the positional board games.

5.1 Move Selection

The major operation of a positional board game program is the one that finds a move for a player. We call **move** the identification and occupation by a player of a position **p** in **E**, where **E** is the set of **empty** positions in a board configuration **C**. Our goal is to select the position p in such a way that the configuration that results by occupying **p** be closer to win than the configuration C, for the player occupying **p**. The library function **find_a_move** performs this operation. The function **find_a_move** uses the search algorithm **alpha-beta** to generate a **partial game tree** whose root node is the current configuration C. A brief description of the search algorithm **alpha-beta** follows.

5.2 The Search Algorithm *Alpha-Beta*

We search a portion of the **complete game tree** called **partial game tree**. We use the **alpha-beta** pruning algorithm to determine the minimax value of the root node (current board configuration) of the partial game tree below this node. The partial game tree is generated in a depth-first manner, omitting all those nodes that do not influence the minimax value of the root node [Cambell 83, Feigenbaum 81, Nilsson 80, Pearl 84]. The **alpha-beta** algorithm shown on figure 5.1 comes from [Cambell 83].

In Figure 5.1, the parameter **c** denotes the **board configuration**, the boolean function **terminal(c)** employs a **termination** criterion to determine whether or not to generate the successor configurations of **c**. It returns **TRUE** when the configuration **c** corresponds to a terminal node (tip-node) of the partial game tree generated. We have mentioned before that it is not computationally feasible to create the complete game tree. So in a partial the game tree, a terminal node may not be truly a terminal node (win or draw) according to the rules of the game. Certain criteria are applied to determine when a node will be considered terminal. With this definition of a terminal node, we need a static evaluation function, **evaluate**, to compute an estimate value of the tip node, i.e. an approximation of the theoretical value that the node would get if complete minimax evaluation were performed. Finally, the function **generate(c)** computes, in some order, all the nodes that correspond to successor configurations of **c**.

5.3 Heuristics in Alpha-Beta

To use efficiently the **alpha-beta** search algorithm we need to provide: (a) a static evaluation function for board configurations, (b) a criterion that will allow us to reorder the successor configurations of any configuration, (c) a pruning criterion that will allow us to prune successor configurations beyond those pruned by the algorithm **alpha-beta**, and (d) a termination criterion that will allow us to determine when to stop expanding the partial game tree below some node.

A **Perfect static evaluation function** reports whether a game tree node leads to a win or a draw. Such a function is not known for the games we treat in this work. If there was such a function there would be no need for searching the game tree below the current configuration node and perhaps no interest in the game. The actual static evaluation function **evaluate** returns an estimate of the value of a configuration node based on the concepts of the **distance to win** and the **promise of a position** which are defined later. There are situations in a play where we cannot have a reliable estimation of the value of a configuration based on the previously described static features. These configurations are called **live** and the condition **liveness**. We subsequently propose: (a) a static evaluation function for the whole class of the positional board games, and (b) a criterion to identify **node liveness**.

The efficiency of all the tree searching methods is usually measured by the number of tip nodes evaluated. The **alpha-beta** search algorithm with

optimal successor-nodes reordering evaluates approximately twice the square root of the tip nodes that the **minimax** method would evaluate. Optimal successor nodes ordering means that the successor node expanded first -in a depth first manner- is the one that will have the best value among the successor configuration nodes. In this case the number of *cutoffs* is maximized. If we knew which node has the greatest theoretical value we would not need to consider all the rest nodes. Usually, we reorder the successor nodes according to a heuristic criterion that attempts to approximate the optimal configuration node reordering. The proposed reordering criterion is presented in section 5.7 in the form of the mathematical relation **LE**.

When the number of successor nodes is large and we suspect that the corresponding values of the successor nodes are such that very few cutoffs occur , like in the game of Go-Moku, we apply a pruning criterion to the successor configuration nodes to ignore some of them and prune the game tree below them. We propose a pruning criterion based on the reordering criterion and the concept of **dead** position.

Finally, the termination criterion controls the size of the partial game tree by determining which nodes, other than terminal configuration nodes, are considered as **tip-nodes**. It is based on the concept of the **nominal maximum depth** of the partial game tree generated, on the **terminal configuration** nodes and on the **live configuration nodes**. The game tree is generated up to a predetermined depth unless a termination configuration is encountered before. This criterion of maximum-depth is bypassed when

a node meets the liveness criterion. In such a case we extend the search until we reach an non-live configuration node. This approach partly compensates for the evaluation errors introduced by the static evaluation function.

In section 5.4, we introduce the concepts of: (a) the "**distance to** the win" of a win group w, (b) **dead** positions, and (c) **configuration liveness**. These concepts apply to any given configuration C of a positional board game. The above concepts form the heuristic basis of the static evaluation function and the reordering criterion.

5.4 Major concepts

In this section we define the concepts of: (a) "**distance** from the win" of a win group, (b) **dead** position in a configuration, and (c) configuration **liveness**.

Let X be in {A, B}. Throughout this chapter when we discuss about a configuration, by convention, we denote with X the player whose turn is to move at that configuration. Y denotes the opponent of X.

We now introduce the concept of "**distance** from the win" for a win group:

(a) Definition of "distance from the win" of a win group

The "distance from the win" of a win group is a function **d** that

takes a board configuration C and a win group w, **open** for player X, as arguments. $d_X(C, w)$ counts the number of holes in w.

In any given configuration C the function $\mathbf{d_X}$ is a lower bound of the number of moves player X has to make before wining using w. It is a lower bound because player X can claim a win by using w if:
(a) player X occupies all the holes in w during his immediately next $d_X(C, w)$ moves and (b) player Y does not occupy any of the holes in w during his immediately next $d_X(C, w) - 1$ moves and (c) none of the immediately following $2*d_X(C, w) - 1$ configurations is a terminal one.

We now introduce the concept of the **dead** positions:

(b) Definition of a dead position

An **empty** position **p**, where **p** is in **P**, is **dead** in a configuration C if p does not belong to any win group **open** for player X or/and player Y in C. We call **DE** the set of **dead** positions in a configuration **C**.

We now introduce the concept of **liveness** for a configuration:

(c) Configuration Liveness

Configuration liveness is a condition of a board configuration. It is associated with the proposed static evaluation function **evaluate**. A **live** configuration node that is also a tip node is not evaluated because the proposed static evaluation function cannot give an accurate estimation for such a configuration node. A board configuration C, where the tree

node corresponding to C is a tip-node of the generated partial game tree, is considered **alive** if there is a win group w, **open** for either player X or player Y, whose **distance** from the win is equal to 1, in C. This implies that there is single **hole** in w. An **alive** configuration node has its successor configuration nodes always generated. This improves the accuracy of the value assigned to configuration C. We discuss how this is achieved in the section that describes the static evaluation function since we need to know how the evaluation function computes the value of a configuration.

5.5 Termination Criterion of the Alpha-Beta Algorithm

We have formulated a search termination criterion common to all the positional board games. It is based on: (a) the nominal maximum depth limit, (b) the condition game_over and (c) the node **liveness**.

First, the function checks whether or not the current configuration is a terminal. If the current configuration is a terminal configuration then the function **continue_search** returns the value **FALSE** otherwise it checks whether the current depth is less than a predefined maximum depth. If the current depth is less than the predefined maximum depth the function returns the value **TRUE** otherwise it checks whether the current board configuration meets the **liveness** criterion. If the current board configuration meets the liveness criterion then the value **TRUE** is returned to the caller otherwise the value **FALSE** is returned and the expansion of the partial game tree terminates below the current board configuration node. This node is considered a tip node and the evaluation function

evaluate is called to compute its value.

5.6 The static evaluation function evaluate

Scope

The static evaluation function **evaluate** computes the value of a board configuration. We use the same static evaluation function for all the positional board games. The computation of the value of a configuration by the function **evaluate** is based on the **promise** of a position. The evaluation function is the algebraic sum of two terms, one for each player. Each term is a function **S** of the all the positions that are holes in open win groups for that player. The difference of these two terms is the value of the configuration. Function S takes one arguments, the configuration C.

In order to define the static evaluation function **evaluate** we need to define the function **promise of a position** which is denoted **PR**. We also need to define one more function, denoted by **S**, which computes the **promise of a configuration**. We define the functions **PR** and **S** separately for each player.

Definition of the function PR - promise of a position

Let CS be the set of all the board configurations and **C** be in CS. Let E_X be the set of **empty** positions in **C** that are holes in an open win group for player X in C and **p** be in E_X. Let $owgx_p = \{ w | w$ is a win group for X and w is **open** for X and **p** is in w$\}$ and let w_{min} be in $owgx_p$ such that

$d_X(C, \mathbf{w_{min}}) <= d_X(C, w)$ for every w in $\mathbf{owgx_p}$. Let **N** be the set of natural numbers. The function $\mathbf{PR_X}$: **CS x $\mathbf{E_X}$ -> N** is defined by:

$$\mathbf{PR_X(C, p) = d_X(C, w_{min})}.$$

The function **PR** measures the *"goodness"* of the position **p** in the configuration C for player X. Let p_i, p_j be in $\mathbf{E_X}$. We consider that position p_i is *"better"* than position p_j if $PR_X(C, p_i) < PR_X(C, p_j)$. Function PR is a heuristic estimation of the value of a position in E_X. Using that value, we are able to compare the any two positions in $\mathbf{E_X}$.

Heuristic basis of the function S

We now describe the heuristic basis of the function S.

Function S is the evaluation of all the **empty** positions that are contained in some win group, **open** for some player X in some configuration C. We group these positions into sets. Any such set contains the **holes** that belong to an **open** win group for player X and have the same promise, in C. $E_{k,C,X}$ denotes the above set of **holes** for player X that have promise equal to k in C. $T_{k, C}$ is the contribution to the value of S of the set $E_{k,C,X}$. The sum of the contributions of the all these sets in a configuration is the value of S in that configuration for player X. $T_{k, C}$ is the sum of the cardinality of the set $E_{k,C,X}$ and the weight $|P|*(2^{mwgx-k} -1)$, where mwgx is the maximum of the sizes of the win groups for player X. The use of the weight allows the contribution of the set $E_{k,C,X}$ to be greater than the sum of the contributions of the all the sets of **empty** positions with promise greater than k for player X in C. We add the above

weight to the cardinality of the set instead of multiplying it with the cardinality in order to keep then values computed by S small. In this way, the set of the positions with the minimum promise in a configuration is the most significant term of function S for player X in C. The use of the cardinality of the set allows us to distinguish between configurations whose positions of minimum promise have the same promise value.

Definition of functions S_X and S_Y

Let CS be the set of all board configuration and C be in **CS**. Let $E_{k, c, x}$ be the set of **empty** positions in C whose **promise** is equal to k. Let mwgx be the maximum of the sizes of the win groups for player X. Let

$$T_{k, x} = \begin{cases} |E_{k, c, x}| + |P|*(2^{mwgx-k}-1), & \text{if } |E_{k, c, x}| <> 0 \\ 0, & \text{if } |E_{k, c, x}| = 0. \end{cases}$$

It can be shown that for any t in {1, ..., mwgx-1}, if $T_{t, x} <> 0$ then $T_{t, x} > |E_{mwgx, c, x}| \quad \sum_{k=t+1}^{mwgx-1} \quad T_{k, x}$.

The function S_X: CS -> N is defined by:

$$S_X(C) = |E_{mwgx, c, x}| + \sum_{k=mhx}^{mwgx-1} T_{k, x},$$

where mwgx is the maximum of the sizes of the win groups for player X and mhx is the minimum **promise** that a position, where p is in **E**, has in **C** for

player X.

Let C_s and C_t be board configurations and X be in {A, B} and E_s be the set of **empty** positions in the configuration C_s and E_t be the set of **empty** positions in the configuration C_t. Let the position p_s be in E_s and let the positions p_t be in E_t such that $PR_X(p_s) <= PR_X(p_{si})$ for every p_{si} in E_s and $PR_X(p_t) <= PR_X(p_{tj})$ for every p_{tj} in E_t. Let S_{Xs} be the sum S_X in the configuration C_s and S_{Xt} be the sum S_X in the configuration C_t for player X. It can be proved that if $PR_X(p_s) < PR_X(p_t)$ then $S_{Xs} > S_{Xt}$ because of the property of the terms $T_{k,X}$, contribution of set $E_{k,C,X}$ of the function S.

Similarly we define S_Y as:

$$S_Y(C) = |E_{mwgy,\ C,\ Y}| + \sum_{k-mhy}^{mwgy-1} T_{k,\ Y},$$

where mwgy is the maximum of the sizes of the win groups for player Y and mhy is the minimum **promise** that a position, where p is in **E**, has in **C** for player Y.

Definition of the function evaluate

Let **CS** be the set of all board configurations that can appear in a positional board game and let **C** be in **CS** and **Z** be the set of integers. The function **evaluate**:**CS** -> **Z** is defined by:

$$\textbf{evaluate}(C) = S_X(C) - S_Y(C)$$

Shortcomings of the static evaluation function and configuration liveness

We now discuss the cases in which the function **evaluate** gives an inaccurate estimation of the value of a configuration. Let C be an **alive** board configuration and w be an **open** win group in C such that **distance** d(C, **w**) = 1, for some player. Let p be the single hole in w and C_b be the configuration that results from C if position p is occupied. There are two cases:

Case 1: The open win group w belongs to player X.
In this case player X has an advantage over player Y which is reflected by the high value of the function S_X. If player Y has also an open win group whose distance from the win is equal to 1 then the function S_Y has also a high value. This implies that the value of the configuration C for player X S_X-S_Y is small which does not reflect the fact that player X can win the game in his next move. Because the evaluation function does not take into account the fact that player Y does not move in C. In order to compensate for that situation we must generate the successor configurations of C.

Case 2: The open win group w belongs to player Y.
The value that the function S_Y gives for C is high. Showing that player Y is in a good game situation. Let us assume that player X has not an open win group whose distance from the win is less than 2 in C. This implies that the value of function S_X is less than the value of the function S_Y. So the value of the configuration C for player X is negative and reflects the fact that the situation for player X is not good in C.

Again, the evaluation function does not take into account that player X moves in C and can close the win group w of Y by occupying the single hole in it. So we must generate the successor configurations of C to anticipate that situation. Note that there might be more than one positions whose promise is equal to 1 for player Y in C. An evaluation function that would take into account all the possible special cases, about open win groups and their holes, would be complicated and slow to compute. By extending the search below alive configurations we can resolve the above special cases.

Now that we have defined the static evaluation function we can prove the following proposition concerning **dead** positions:

Proposition 4

If C_s is a board configuration and **DE** is the set of **dead** positions in C_s and p is in **DE** and C_t is the board configuration that results from the configuration C_s by occupying position p then **evaluate**(C_s) = **evaluate**(C_t).

Proof

Let E′ be the set of the **empty**, **non-dead** positions in configuration C_s. Since p is not in E′, the set of **empty**, **non-dead** positions in configuration C_t is the same as in the configurations C_s, namely E′. The win groups that are open for player X or/and player Y in the configuration C_s are also open, for the same players, in the configuration C_t. Note that the number of **holes** of each one of those **open** win groups is the same in configurations C_s and C_t which implies that the promise of

every position in E′ is the same in configurations C_s and C_t which in turn implies that $S_X(C_s) = S_X(C_t)$ and $S_Y(C_s) = S_Y(C_t)$. So **evaluate**(C_s) = evaluate(C_t).

5.7 Reordering Criterion

Scope of the Relation LE

The relation **LE** is a heuristic criterion that allows us to determine the order in which we generate the successor configurations of some configuration. This is called **plausibility ordering** and its purpose is to improve the efficiency of the **alpha-beta** search algorithm - approximation of the optimal ordering. It is used to compare two **empty** positions that are not **dead** in any given configuration C. If C is a board configuration, E is the set of **empty** positions in C and **DE** is the set of dead positions in C then **LE** determines a total order among the elements of **E-DE**. The basis of the reordering criterion is the minimum promise of a position for any of the two players. We call that minimum promise **TPR**.

In order to define the relation **LE** we first need to define two functions called **TPR** and **WTPR**. The relation **LE** is defined in terms of these two functions.

The function **TPR** is the minimum promise of a position in some configuration C. The minimum promise of a position is independent of player.

Function TPR

Let **CS** be the set of all board configurations and **C** be in **CS**. Let E be the set of **empty** positions in **C** and p be in **E** and **P** = {1, ..., |P|}. the function **TPR**: **CSxE** -> **P** is defined by:

$$TPR(C, p) = \min \{PR_X(C, p), PR_Y(C, p)\}$$

Function WTPR

The function **WTPR** takes a configuration C and a position p in P as arguments. **WTPR(C, p)** counts the number of open win group of both players whose distance to win is equal to TPR(C, p).

Definition of Relation LE

Let C be a board configuration and E be the set of **empty** positions in C and DE be the set of **dead** positions in C. We define the binary relation **LE** on E-DE as: $\{(p_i, p_j) |$ if $(TPR(p_i) < TPR(p_j))$ or $(TPR(p_i) = TPR(p_j)$ and $WTPR(p_i) >= WTPR(p_j))$ holds$\}$.

Let p_i, p_j be in E-DE such that (p_i, p_j) is in LE, in configuration C. Let C_i be the successor configuration of C that results by occupying position p_i and C_j be the successor configuration of C that results by occupying position p_j. Let $owg_{i,j}$ be the set of open win groups, for any of players, that contain either position p_i or position p_j, in C. These are the win groups whose state is affected by occupying either p_i or p_j. We generate the successor configuration that results by occupying position p_i before the configuration that results by occupying p_j because we are interested in changing the state of those win groups that have the minimum

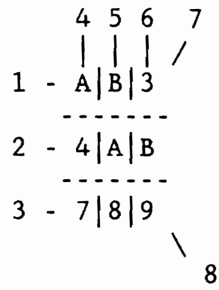
distance from the win.

We generate the configuration C_i before the configuration C_j because there is an open win group in $owg_{i,j}$, which contains position p_i, and its distance from the win is the minimum distance from the win that any win group has in owg.

```
alphabeta(c:configuration;a,b:integer)
{
 m,i,t,w :  integer ;

   if (terminal(c))
       return(evaluate(c)) ;

   w = generate(c) ;              /* determine successors c1, c2, ..., cw*/
   m = a ;

   for i=1 to w do
   {
     t = - alphabeta(ci,-b,-m) ;

     if(t>m)
          m = t ;

     if(m>=b)
        return(m) ;   /* cutoff */

  }
        return(m) ;
```

Figure 5.1

Alpha Beta Game Tree Search Algorithm



win group #:	1	2	3	4	5	6	7	8
win group distance:	-	-	3	2	-	-	2	1

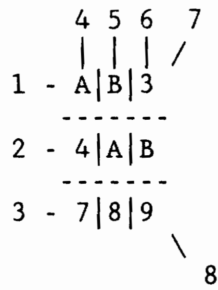
position identifier:	1	2	3	4	5	6	7	8	9
owg_p :	-	-	{7}	{4}	-	-	{3, 4, 7}	{3}	{3, 8}
$PR_X(C, p)$:	-	-	2	2	-	-	2	3	1

k:	1	2	3
$E_{k, C, X}$:	{9}	{3, 4, 7}	{8}
T_k:	$1 + (2^{3-1}-1)*9$	$3 + (2^{3-2}-1)* 9$	1

$S_X(C)$: 1 + 27 + 3 + 9 + 1 = 41

Figure 5.2

Example of the static evaluate function evaluate



win group #:	1	2	3	4	5	6	7	8
Player X, win group distance:	-	-	3	2	-	-	2	1
Player Y, win group distance:	-	-	3	-	-	2	-	-

position identifier:	1	2	3	4	5	6	7	8	9
Player X, owg_p :	-	-	{7}	{4}	-	-	{3, 4, 7}	{3}	{3, 8}
$PR_X(C, p)$:	-	-	2	2	-	-	2	3	1
Player Y, owg_p :	-	-	{6}	-	-	-	{3}	{3}	{3, 6}
$PR_Y(C, p)$:	-	-	3	-	-	-	3	3	2
TPR :	-	-	2	2	-	-	2	3	1

Order of positions in E according to LE: (9, 7, 3, 4, 8)

Figure 5.3

Example of relation LE

CHAPTER 6

IMPLEMENTATION OF THE GAMES OF HEX and GO-MOKU

Abstract

In Chapter 6 we describe the implementation of programs that play the positional board games of Hex and Go-Moku. These programs are based on the library. Furthermore, we discuss which of the modules that constitute a positional board game program vary from one game to another. These are: the package **board** and the procedure **init_screen_coordinates** of the package **User_interface** where the package **board** of each positional board game program is generated by same the program that computes the sets of the win groups of the two players.

6.1 Game Specific Elements of a Positional Board Game Program

In order to build a positional board game program using the library we need to provide: (a) the package **board** of that game and, (b) the body of procedure **init_screen_coordinates** of the package **User_interface**.

Procedure init_screen_coordinates

Chapter 4 describes the scope and the coding details of the procedure **init_screen_coordinates**. We simply mention here that this procedure is declared as a **separate** procedure in the body of the package User_Interface. The library may contain more than one procedure with the

same name **init_screen_coordinates** where each one corresponds to a different positional board game program and must reside in a separate file. Each implementation of the procedure **init_screen_coordinates** must provide the values of the data objects: **b_ch**, **total_lines**, **total_columns**, **xy_display**, **background**, and **xy_background** which depend on the physical board of each positional board game. The previous data objects are described in Chapter 4.

Figure 6.1 shows the procedure **init_screen_coordinates** corresponding to the program that plays the game of Hex while Figure 6.2 shows the procedure **init_screen_coordinates** corresponding to the program that plays the game of Go-Moku.

Package board

We need to build the package **board** for each positional board game implemented. The program entities contained in this package have been described in detail in Chapter 3. The variable parts of this package are the values of the constants: **number_of_positions**, **wga**, and **wgb** which contain the abstract **board** characteristics.

We have coded a program for each game that computes the sets of the win groups and generates (writes) the corresponding package **board**. The major elements of such a program are: (a) an enumeration scheme that

assigns a distinct number **pid** in {1, ..., |P|} to each position of the physical board (b) the cardinality |P| of the set P and (c) the computation of the sets of the **useful** win groups from the procedural game rules that describe all the groups of the game. The scope of the enumeration of the physical board positions is to allow us to identify each one of them uniquely in a win group. The cardinality |P| of the set P is derived by the definition of the physical board of the positional board game implemented. The form of the output of the program that generates the package **board** is described in Chapter 3. As an example, Appendix C contains the package **board** of each of the two games.

Enumeration of the board positions of a two-dimensional physical board

Using the library we are able to display physical boards of positional board games that can be represented as two-dimensional NxM arrays. Such a physical board is enumerated row-wise from top to bottom and from left to right. A position p that is located at line rp and column cp of a physical board represented by a two dimensional array is assigned the number

pid = N*(rp-1) + cp.

By building and compiling the above two modules we have in place all the parts of a new positional board game. The user of the library does not need to modify any program entity inside any of the other packages.

6.2 Game of Go-Moku: Game Definition and Win Group Computation Algorithm

6.2.1 Definition of the positional board game of Go-Moku

Definition of the physical board

The physical board of the game of Go-Moku is a two dimensional 19x19 array. Each array component represents a board position. The cardinality of set **P** is 361. Figure 6.3 shows an enumerated physical representation of a Go-Moku board.

Procedural definition of the win groups

Any 5 consecutive aligned positions -horizontally, vertically, or diagonally- form a win group. The sets **wga** and **wgb** of the win groups are the same. The previous rule defines only **useful** win groups.

Derivative Go-Moku games

We can obtain derivative versions of the game of Go-Moku by changing the size of the physical board. Specifically by redefining the dimensions of the board from 19x19 to NxN, where N denotes both the number of the lines and the number of the columns. The game rule that defines the win groups remains the same.

We have implemented the game of Go-Moku played on a board 7x7. We have implemented the game Go-Moku in a smaller size board in order to

reduce the branching factor of the **alpha-beta** search algorithm and increase the program speed during the computation of a move for a player. The branching factor of a Go-Moku game is not a constant number. In each board configuration the branching factor is equal to the number of the **empty** positions in that configuration. So during the early stages of a play the branching factor is equal to several hundreds. The Go-Moku board positions have been enumerated according to the enumeration scheme introduced in the previous section.

6.2.2 Win Group Computation Algorithm for the game of Go-Moku

The program **Go_Moku_GWG** generates the sets **wga** and **wgb** of the win groups and the corresponding package **board** for a Go-Moku game with board dimensions NxN where N is provided by the user. The listing of the program **Go_Moku** can be found in the Appendix.

Cardinality of the sets wga and wgb

Each line of a Go-Moku NxN board has **N-4** distinct win groups since the leftmost position in a chain of five consecutive positions has to be within either the first N-4 positions in a line or the last N-4 positions in a line. Similarly each column has **N-4** distinct win groups. The total number of distinct win groups that reside at the lines and columns of an NxN Go-Moku board is **2*N*(N-4)**.

Furthermore, each one of the N-4 leftmost positions of the N-4 top lines of an NxN Go-Moku board is the topmost position of a distinct left-to-right diagonal chain of five consecutive positions (win-group). Since there are (N-4)*(N-4) such positions there are also (N-4)*(N-4) win groups that are on left-to-right diagonals. Similarly, each one of the N-4 rightmost positions of the N-4 top lines of an NxN Go-Moku board is the topmost position of a distinct right-to-left diagonal chain of five consecutive positions (win-group). Since there are (N-4)*(N-4) such positions there are also (N-4)*(N-4) win groups that are on right-to-left diagonals. So there are **2*(N-4)*(N-4)** win groups that are on all the diagonals of an NxN Go-Moku board. The total number of win groups in an NxN Go-Moku board is

$$2*N*(N-4) + 2*(N-4)*(N-4).$$

Basis of the algorithm

The algorithm that creates the sets of the win groups for an NxN Go-Moku board is shown at Figure 6.4, where $p_{l,c}$ denotes the identification number assigned to that position by the enumeration scheme as a function of l and c. The position $p_{l,c}$ is located at line l, column c of the board. The computation of the win groups is accomplished as follows: we visit each Go-Moku board position $p_{l,c}$ that can be at the end of a chain c of five consecutive positions in any direction-horizontal, vertical, or diagonal. For each of the four directions, we compute the coordinates of the remaining 4 positions of the chain c and for each of them we compute

the identification number according to the enumeration scheme. We use these identification numbers to form the win group according to the model.

6.3 Game of Hex: Game Definition and Win Group Computation

6.3.1 Definition of the positional board game of Hex

Game Definition

The game of Hex is played on a diamond shaped region (physical board) where each board position is physically represented by a **hexagon** and each side of the region is 11 positions long. The number of the board positions (**hexagons**) is equal to 121.

An 11x11 Hex physical board can be represented as a two dimensional 11x11 orthogonal array where each position is mapped into an array component. This component is connected by lines to other array cell that correspond to adjacent hexagons in the original board preserving in this way the connectivity properties of the hexagons. Figure 6.5 shows the equivalent two-dimensional Hex board which will be used from further on. We have named the sides of an Hex board as follows: the top side is called North (N), the bottom side is called South (S), the left side is called West (W), and the right side is called East (E).

In order to define the sets of the win groups we first need to define what a **chain** of positions in a Hex board is.

Definition of a chain of positions as a win group

A **chain** c is a sequence of positions that connects two opposite sides of a Hex board. A chain is denoted by (p_1 , p_2, ..., p_k), where p_1 is on one side of the Hex board, p_k is on the opposite side and, for every s in {1, ..., k-1} p_s and p_{s+1} are physically adjacent Hex board positions.

Definition of the sets of the win groups

Any chain of positions that connects two opposite sides of the board constitutes a win group. Specifically, a chain (p_1 , p_2, ..., p_k) corresponds to the win group {p_1 , p_2, ..., p_k} and for every win group {p_1 , p_2, ..., p_k} there exists a permutation of its positions that is a chain. This definition of the win groups includes useful, usable and useless win groups. In section 6.3.2 we present the conditions that allow us to identify the useful win groups.

As a matter of convention, we assume that player A must connect the North with the South side of the board while player B must connect the West with the East side of the board.

Derivative games of the game Hex

Derivations of the game of Hex can be obtained by replacing the number of the positions at each side from 11 to N, where N >= 3. The rule that defines the sets of the win groups is the same. We have implemented the game of Hex with 4 (Hex/4) and 5 (Hex/5) positions at each side of the board. Similar to the game of Go-Moku, program speed is the reason behind the decision to implement the game of Hex on a smaller size board.

6.3.2 Identification of usable and useful win groups

Since the definition of the sets of the win groups contains **useful**, **usable**, and **useless** win groups we need a way to distinguish the **useful** win groups from the **useless** and **usable** win groups. From the model of [Antoy 87] we know that the **useful** win groups are those that are important for game-playing purposes. Since the definition of the win groups for the game of Hex contains a very large number of win groups, by incorporating only the **useful** win groups-a fraction of the total number of win groups-in a program, we avoid a substantial performance degradation of the program.

We need to identify: (a) which win groups are **useless** , and (b) which win groups are **usable** but not **useful**. Condition 1 below allows us to identify the **useless** win groups and Condition 2 allows us to identify which of the **usable** win groups are also **useful** win groups.

Condition 1: Identification of the useless win groups

Let **c** be a chain of positions on an NxN Hex board and let **k** be the number of positions in the chain **c**. Let **u** be equal to **N*N/2**, if N is even or **(N*N+1)/2**, if N is odd. Chain **c** denotes a **usable** win group iff

$$k <= u.$$

Proof

The proof of the Condition 1 is easily derived by Theorem 3 in [Antoy 87].

Condition 2: Identification of the useful win group

Let us assume that Condition 1 holds for the chain $c=(p_1, p_2, ..., p_k)$ which corresponds to the **usable** win group $w = \{p_1, p_2, ..., p_k\}$. Let us call A and B the sides connected by c. The win group w is **useful** iff:

(a) p_1 is the only position in c that is on the side A and

(b) p_k is the only position in c that is on the side B and

(c) position p_1 is adjacent only to position p_2 in the chain c and

(d) position p_k is adjacent only to positions p_{k-1} in the chain c and

(e) position p_n is adjacent only to positions p_{n-1} and p_{n+1} in the chain c for every n in $\{2, ..., k-1\}$

Conditions 2(a) and 2(c) demand that position p_1 is the only position of chain c that is on side A of the board and the only other position of chain c that is adjacent to p_1 is position p_2.

Conditions 2(b) and 2(d) demand that position p_k is the only position of chain c that is on side B of the board and the only other position of chain c that is adjacent to p_k is position p_{k-1}.

Condition 2(e) demands that every position p_n of chain c, where n is in {2, ..., k-1}, has only two other adjacent positions in c, the preceding (p_{n-1}) and the following (p_{n+1}) ones.

Now we show that if any of the Conditions 2(a)-2(e) does not hold for the chain c = (p_1 , p_2, ..., p_k) then we can derive a subchain of c that still connects A with B. A subchain cs of a chain c connects the same two opposite side of the Hex board as chain c and every position p_s in chain cs is also in chain c and chain cs contains only some of the positions p_s in chain c and the order of the positions in the two chains is the same. Let w denote the win group corresponding to chain c and ws denote the win group corresponding to chain cs. The win group w contains the win group ws so w is not a **useful** win group.

Proof

Necessity of (a)

If Condition 2(a) does not hold for the chain c=(p_1 , p_2, ..., p_k) then there is a position p_n in chain c on side N, where n is in {2, ..., k-1}. If c is a chain, then

$$cs = (p_n , p_{n+1}, \dots, p_k)$$

is a chain that connects the same two opposite sides of the Hex board as the chain c. The positions $p_1, \ldots, p_{n-1}$ of the chain c are not in chain cs. If w = $\{p_1, p_2, \ldots, p_k\}$ is the corresponding win group of the chain c and ws = $\{p_n, p_{n+1}, \ldots, p_k\}$ is the corresponding win group of the chain cs then ws is a proper subset of w since w - ws = $\{p_1, \ldots, p_{n-1}\}$, where w - ws $<> \emptyset$. So w is not a **useful** win group.

Necessity of (b)

The proof of Condition 2(b) is symmetric to the proof of the Condition 2(a).

Necessity of (c)

If condition 2(c) does not hold for the chain c=$(p_1, p_2, \ldots, p_k)$ then there is a position p_n in chain c adjacent to position p_1, where n is in $\{3, \ldots, k-1\}$. If c is a chain, then

$$cs = (p_1, p_n, \ldots, p_k)$$

is a chain that connects the same two opposite sides of the Hex board as the chain c. The positions $p_2, \ldots, p_{n-1}$ of the chain c are not in chain cs. If w = $\{p_1, p_2, \ldots, p_k\}$ is the corresponding win group of the chain c and ws = $\{p_1, p_n, \ldots, p_k\}$ is the corresponding win group of the chain cs then ws is a proper subset of w since w - ws = $\{p_2, \ldots, p_{n-1}\}$, where w - ws $<> \emptyset$. So w is not a **useful** win group.

Necessity of (d)

The proof of Condition 2(d) is symmetric to the proof of the Condition 2(c).

Necessity of (e)

If condition 2(e) does not hold for the chain $c=(p_1, p_2, \ldots, p_k)$ then for some position p_n in c, where n is in {2, ..., k-1}, there is position p_m in c, where m is {2, ..., k-1} - {n-1, n+1}, adjacent to position p_n. There are two cases for position p_m:

Case 1: position p_m precedes position p_n in chain c where m is in {2, ..., n-2}

If m is in {2, ..., n-2} then the chain

$$cs = (p_1, \ldots, p_m, p_n, \ldots, p_k)$$

still connects the same two opposite sides of the Hex board as the chain c. The positions $p_{m+1}, \ldots, p_{n-1}$ of the chain c are not in chain cs. If $w = \{p_1, p_2, \ldots, p_k\}$ is the corresponding win group of the chain c and $ws = \{p_1, \ldots, p_m, p_n, \ldots, p_k\}$ is the corresponding win group of the chain cs then ws is a proper subset of w since $w - ws = \{p_{m+1}, \ldots, p_{m-1}\}$, where $w - ws \neq \emptyset$. So w is not a useful win group.

Case 2: position p_m follows position p_n in chain c where m is in {n+2, ..., k-1}

The proof of Case 2 is symmetric to the proof of Case 1.

Sufficiency of Condition 2

We want to prove that if Condition 2 holds for chain c and w is the win group corresponding to c then w is a **useful** win group. We prove this by showing that if w is not a **useful** win group then Condition 2 does not hold for chain c.

Let us assume that the win group w that corresponds to the chain c = $(p_1, \ldots, p_k)$ is not **useful**. This implies that there exists a win group ws which is a proper subset of w. Let p_t, where t is $\{1, \ldots, k\}$ be in w and not in ws. Without loss of the generality we select p_t such that there exists position p_a in ws adjacent to p_t. Notice that position p_a is also in w.

There are two cases for position p_a:

Case 1: position p_a is on either one of the two sides of the board in this case

(a) position p_a has one adjacent position in ws

and

(b) position p_a has one adjacent position in w

Since position p_t is not in ws, position p_a has at least two adjacent positions in w which implies that either of Conditions 2(c) or 2(d) does not hold.

Case 2: position p_a is not on either one of the two sides of the board

In this case

(a) position p_a has two adjacent positions in ws

and

(b) position p_a has two adjacent positions in w.

Since position p_t is not in ws, position p_a has at least three adjacent positions in w which implies that Condition 2(e) does not hold.

6.3.3 Win Group Generation for the game of Hex

The program **Hex_GWG** generates the sets of the **useful** win groups and the package **board** that corresponds to the game of Hex with N positions at each side of the board. More specifically, program **Hex** generates all the chains of positions that connect the top with the bottom side (the corresponding win groups are in wga) and the left with the right side (the corresponding win groups are in wgb) of an NxN Hex board. Conditions 1 and 2 are used by the program to determine whether a win group is **useful** or not.

Description of the algorithm

Let us assume that we want to generate all the chains of positions that connect the North side with the South side of an NxN Hex board. Let N be the set of positions on side North and S be the set of positions on

side South. For each position p in N we build the tree of all the chains of positions whose first position is p and their last position is in S. The root node of the tree denotes the position p and the children of a node denoting any position q in the tree denote all positions adjacent to q in the NxN Hex board, which do not appear in the path from the root p to q. The leaves of the tree with root node p correspond to board positions on South side . A path from the root to a leaf that satisfies Conditions 1 and 2 corresponds to a **useful** win group. The tree is generated in a depth-first manner. We generate each one of the children nodes of a tree node we check if the path from the root to newly added node satisfies Conditions 1 and 2. If the path with a newly added node does not satisfy these two conditions the last node added to the path is discarded and some other node that corresponds to an adjacent position to the parent node is tried.

The main algorithm that computes the chains of positions that correspond to **useful** win groups is shown at Figure 6.6. The program can be found at the Appendix A. In Figure 6.6:

function **neighbors(p)** computes all the positions that are adjacent to position **p** in Hex board and returns them to the caller.

function **useless(d)** determines whether the current chain will correspond to a **useless** win group when completed (Condition 1). If Condition 1 holds it returns the value **TRUE** otherwise it returns the value **FALSE**. Parameter **d** denotes to the number of the positions already in the chain.

function **useful(c, d)** checks whether Condition 2 holds for the chain **c**. If Condition 2 holds it returns the value **TRUE** otherwise it returns the value **FALSE**. Parameter **d** denotes to the number of the positions already in the chain.

```
separate(user_interface)

procedure init_display_coordinates is
begin
  b_c := 56;
  line := 5;
  column := 5;
  enum := 1;
  FOR lines IN 1 .. line LOOP
    FOR columns IN 1 .. column LOOP
      pos_dir(lines, columns) := enum ;
       enum := + 1;
    END LOOP;
  END LOOP;
  xy_display := ((4, 6), (4, 8), (4, 10), (4, 12), (4, 14), (6, 6), (6,
8), (6, 10),(6, 12), (6, 14), (8, 6), (8, 8), (8, 10), (8, 12), (8, 14),
(10, 6), (10, 8), (10, 10), (10, 12), (10, 14), (12 ,6), (12, 8), (12,
10), (12,12), (12, 14));

  xy_background(1 .. 56) := ((4, 7), (4, 9), (4, 11), (4,13), (5, 6), (5,
7), (5, 8), (5, 9), (5, 10), (5, 11), (5, 12), (5, 13), (5, 14), (6, 7),
(6, 9), (6, 11), (6, 13), (7, 6), (7, 7), (7, 8), (7, 9), (7, 10), (7,
11), (7, 12), (7,13), (7, 14), (8, 7), (8, 9), (8, 11), (8, 13), (9, 6),
(9, 7), (9, 8), (9, 9), (9, 10), (9, 11), (9, 12), (9, 13), (9,14), (10,
7), (10, 9), (10, 11), (10, 13), (10, 14), (11, 7), (11, 8), (11,  9),
(11,10), (11,11), (11,12), (11,13), (11,14), (12, 7), (12, 9), (12, 11),
(12, 13)) ;

  background(1 .. 56) := ('-', '-', '-', '-', '|', '/', '|', '/', '|',
'/', '|','/', '|','-', '-', '-', '-', '|', '/', '|', '/', '|', '/',
'|','/', '|','-', '-', '-', '-', '|', '/', '|', '/', '|', '/', '|','/',
'|','-', '-', '-', '-', '|', '/', '|', '/', '|', '/', '|','/', '|','-',
'-', '-', '-');

end init_display_coordinates ;
```

Figure 6.1

Procedure init_display_coordinates for the game of Hex (5x5)

```
separate(user_interface)

procedure init_display_coordinates is
lm : natural := 5 ;
cm : natural := 5 ;
begin
  b_c := 0 ;
  line := 7;
  column := 7;
   := 1;
  FOR lines IN 1 .. line LOOP
    FOR columns IN 1 .. column LOOP
      pos_dir(lines, columns) := ;
        :=  + 1;
    END LOOP;
  END LOOP;
  for lines in  1 .. line loop
     cm := 5 ;
    for columns in 1.. column loop
        xy_display(pos_dir(lines,columns)).x := lm  ;
        xy_display(pos_dir(lines,columns)).y := cm + columns ;
        cm := cm + 1 ;
  end loop ;
       lm := lm + 1 ;
end loop ;
end init_display_coordinates ;
```

Figure 6.2

Procedure init_display_coordinates for the game of Go-Moku (7x7)

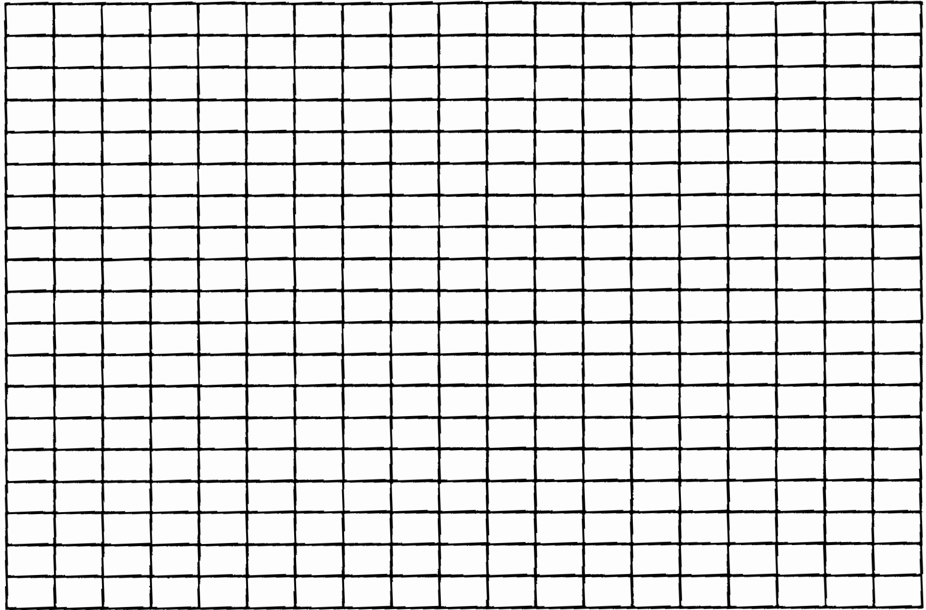


Figure 6.3

Physical board for the 19x19 dimension game Go-Moku

```
counter := 1

For l in 1..N loop
      For c in 1..N-4 loop
       wga(counter) := (p_{l,c}, p_{l,c+1}, p_{l,c+2}, p_{l,c+3}, p_{l,c+4})
       counter := counter + 1
      end loop
end loop

For c in 1..N loop
      For l in 1..N-4 loop
       wga(counter) := (p_{l,c}, p_{l+1,c}, p_{l+2,c}, p_{l+3,c}, p_{l+4,c})
       counter := counter + 1
      end loop
end loop

For l in 1..N-4 loop
      For c in 1..N-4 loop
       wga(counter) := (p_{l,c}, p_{l+1,c+1}, p_{l+2,c+2}, p_{l+3,c+3}, p_{l+4,c+4})
       counter := counter + 1
      end loop
end loop

For l in 1..N-4 loop
      For c in reverse 1..N-4 loop
       wga(counter) := (p_{l,c}, p_{l+1,c-1}, p_{l+2,c-2}, p_{l+3,c-3}, p_{l+4,c-4})
       counter := counter + 1
      end loop
end loop

wgb := wga
```

Figure 6.4

Win group sets Generation algorithm for the game Go-Moku NxN

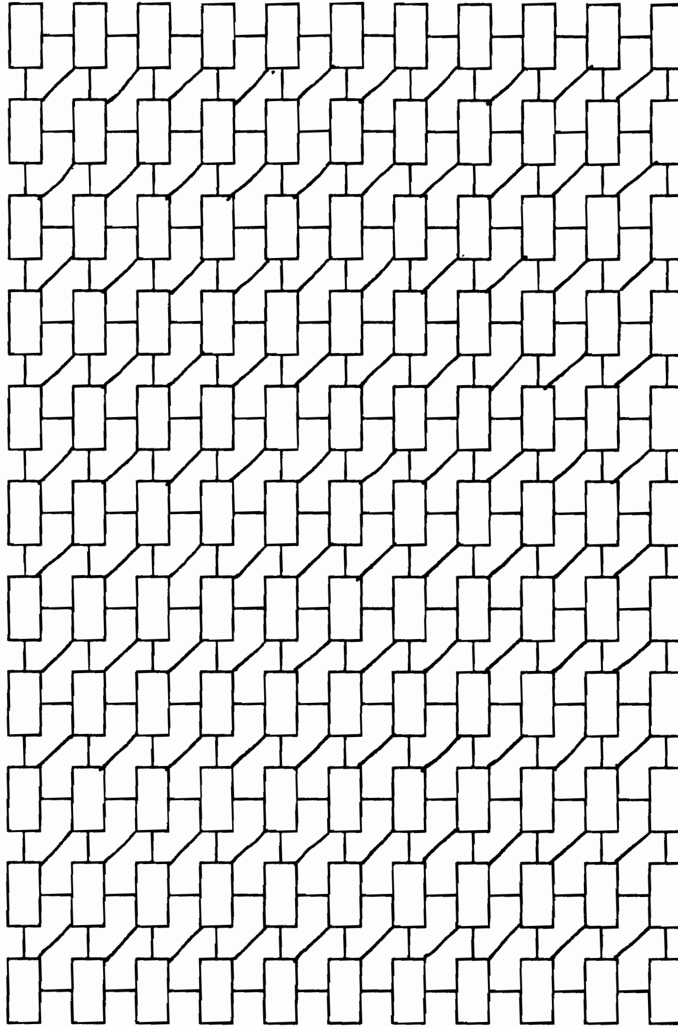


Figure 6.5

Square representation of an 11x11 Hex board

```
 wg_size := 0

 For each position pa in side(A) loop
        depth := 1
        path(depth) := pa
        occupied(pa) := true
        create_paths(pa, depth+1)
        occupied(pa) := false
end loop

procedure create_paths(p,d) is
  begin
        if p in side(B) then
           if not useless(d-1) and not usable(path,d-1) then
                 wg_size := wg_size + 1
                 wg(wg_size) := path(1..d-1)
           end if
       else
         if not useless(d-1) and not usable(path,d-1) then
              s := neighbors(p)
              for p1 in s loop
                 if not occupied(p1) then
                     path(d) := p1
                     occupied(p1) := true
                     create_paths(p1,d+1)
                     occupied(p1) := false
                  end if
              end loop
        end if
     end create_paths
```

Figure 6.6

Win group sets Generation Main Algorithm for the game Hex

CHAPTER 7

PROGRAMMING ISSUES OF THE IMPLEMENTATION

Abstract

This chapter describes the major data types, data objects, and subprograms contained in the library packages and their scope. It also discusses some programming issues that were faced during the implementation phase and the implementation principles.

7.1 Implementation Principles

The program code that implements the subprograms declared in the specification part of each library module consists of small-sized subroutines. Each one of these subroutines carries out a single function and shares as little global data objects as possible with the other subprograms. Minimizing the global data sharing with the use of appropriate parameter passing eliminates the side effects. The use of Ada functions, instead of Ada procedures where possible, allowed us to pass the result of the operation implemented by the function to the caller through a data value returned by the function rather than modifying a global variable or an IN/OUT actual parameter. Furthermore we have used constants and static data structures whenever possible to reduce the execution-time overhead and the possibility of run-time error, due to lack of memory, associated with the creation of dynamic data structures. The program code of all the package bodies is provided in the Appendix A. In the rest of the chapter, we present the major program entities of the

package bodies.

7.2 Implementation of the Package Body Game

General

The package body **game** contains the subprograms that coordinate the play of a positional board game. These subprograms handle the interaction with the user by getting the user selections and executing them. We need to remind that player A who is denoted by the value **player_a** of type **values** plays first by definition.

Data Types and Data Objects

Data type players and data object first_player

The data type **players** is an enumerated data type of two values: **human, computer**. The value **human** denotes the user and the value **computer** denotes the program. The data object **first_player** of type **players** denotes which of the actual player plays first. The player who plays first takes the role of the player A.

Data objects move_counter and move_sequence

The data object **move_counter** is a variable of type **natural** in the range from **0** to **number_of_positions**. It is a counter that keeps the number of the occupied board positions in the current configuration, in a play. If the variable **move_counter** is assigned the value **0** then the play

has not started yet -**new game**. The data object **move_sequence** is an array of size **number_of_positions** where its components are of type natural in the range 1 to **number_of_positions**. The value of the component **move_sequence(i)** denotes the identification number of the board position occupied during move **i**.

Subprograms

Procedure play_game

The procedure **play_game** coordinates the play of a positional board game. It calls subprograms of the package **user_interface** to handle screen input/output, the package **configuration** to access the board configuration and the package **move** to find and make a move.

Specifically, it initializes the display by calling the procedures **init_display** and **display_board** from the package **user_interface** and starts a loop which is terminated by the user. Inside the loop, the procedure **play_game** calls the function **get_user_selection**, which assigns a value of the data type **values** to the variable **selection**, to get the user choice. The variable **selection** is passed as a parameter to the procedure **execute** which, for each value of the data type **values**, calls an appropriate procedure to perform the operation that corresponds to the user choice.

7.3 Implementation of the Package Body Configuration

General

The package body **configuration** contains the data object **board_configuration** which is used to store the **board configuration** and the implementation of the subprograms that provide access to the **board configuration.** The data type and the data object that are used to represent the **board configuration** are kept hidden from the user of the library. Access to the **board configuration** is attained only by calling the subprograms declared in the specification part of the package **configuration.** An array of size $|\mathbf{P}|$ of components of type **values** represents the **board configuration.** The index **i** of the array **board_configuration** ranges from 1 to $|\mathbf{P}|$. The index **i** is the position identification number assigned by the enumeration scheme described in Chapter 6. The operations on the **board configuration** access individual positions. For of this reason, we have chosen an array to represent the **board configuration** as the fastest and most straightforward way.

Data Types and Data Objects

Data type board_configuration_type

The data type **board_configuration_type** is defined as a one dimensional array of size $|\mathbf{P}|$ where each component is of type **values.** It is used to declare data objects that store the **board configuration.**

Data object board_configuration

The data object **board_configuration** is of the data type **board_configuration_type** and is used to store the **board configuration**.

7.4 Implementation of the Package Move

General

The package body **move** contains the implementation of the subprograms **find_a_move** and **make_a_move** and the representation of the **open** win groups. The implementation of the procedure **find_a_move** is based on the **alpha-beta** search algorithm presented in Chapter 5. The procedure **make_a_move** updates the program entities that represent: the **board configuration**, the open win groups and the **current player**. In this section, we present the major data types, data objects and subprograms that are used by the above two procedures.

Data types and data objects

Data objects wg_size_a and wg_size_b

The constants **wg_size_a** and **wg_size_b** are of the data type **integer**. The constant **wg_size_a** is assigned the value **wga'length**, which is side of the array **wga**, and the constant **wg_size_b** is assigned the value **wgb'length**, whic is the size of the array **wgb**.

Data type best_rec

The data type **best_rec** is a record data type with two fields: **value** of type **integer** and **move** of type **board_range**. A data object of this data type is returned by the function **alpha_beta** to its caller. The **alpha_beta** function assigns the computed value of the current board configuration to the field **value** and an **empty** board position to the field **move**. The value stored at field **value** is the **alpha_beta** estimated value of the configuration that results from the current configuration by occupying the position stored at **move**.

Data types slot_type and list

The data type **slot_type** is defined as an one-dimensional array of size |P| of components of the data type **board_range**. The data type **slots_type** is used in the declaration of the data type **list**. The data type **list** is a record with two fields: the field **how_many** of type **integer** in the range from 0 to |P| and the field **slot** of type **slot_type**. Data objects of the data type **list** denote lists of board positions. The number of positions in the list is stored at the field **how_many** and the board positions in the list are stored at the first **how_many** components of the field **slot**.

Data type open_wg_type

The data type **open_wg_type** is a record data type with two fields: the field **open** of the data type **boolean** and the field **open_positions** of

the data type **integer**. The data type **open_wg_type** is used in the declaration of the data objects **free_wga** and **free_wgb**.

Data objects free_wga and free_wgb

The data object **free_wga** is a one dimensional array of size |**wga**| of components of the data type **open_wg_type**. The component **free_wga(wid)** of the data object **free_wga** corresponds to the win group stored at the component **wga(wid)** of the data object **wga** in the package **board**. It denotes the state of the win group stored at **wga(wid)** in the current configuration. The state of any win group for some player in a configuration is determined by two factors: whether the win group is open or not and the number of holes in that win group. The field **free_wga(wid).open** is assigned the value **TRUE** if the win group stored at **wga(wid)** is open and the field **free_wga(wid).open_positions** is used to store the number of holes of the win group stored at **wga(wid).wg**. Similarly, the data object **free_wgb** is an array of size |**wgb**| of components of the data type **open_wg_type**. Each win group stored at a component of **wga** is identified uniquely in the whole library by the array index **wid**, where **wid** is in {1, ..., **wga'length**}. During game playing, the operations on the win groups stored at **wga** are: the access and change of the state of a win group. The array represention of the state of the win groups allows us to access the state of a win group fast and easily through the use of the same identifier **wid** which is used to access the component of the table **wga** where the win group is stored. The same

discussion also applies to the data object **free_wgb** which refer to the state of the win groups stored at **wgb**.

Data objects pos_in_wga and pos_in_wgb

The data objects **pos_in_wga** and **pos_in_wgb** keep track of which win groups contain each board position for each set of win groups. The data object **pos_in_wga** is of type **wg_table(board_range)**. The component **pos_in_wga(p)** of the array **pos_in_wga** corresponds to the board position **p** and is used to store the win group identifiers of all the win groups stored at the data object **wga** that contain position **p**. These are stored at the field **wg** of **pos_in_wga(p)**. The number of the win groups stored at **wga** that contain position p is stored at the field **size** of **pos_in_wga(p)**. The data object **pos_in_wgb** is of the data type **wg_table(board_range)**. It has the same scope as the data object **pos_in_wga** but it refers to the win groups stored at **wgb**. Figure 7.1 shows the contents and the structure of the data object **pos_in_wga** for the game of **Tic-Tac-Toe**. The numbers inside the **Ti-Tac-Toe** board identify the positions and the numbers outside identify the win groups. The scope of these data objects is to allow us to locate fast which win group are affected after the occupation of an **empty** board position **p** without having to search the whole tables **wga** and **wgb**. The change of the value of a position which is not **dead** affects the state of all the win groups of both players that contain that position. Since the change of the value of a board position is a frequent operation during the **alpha-beta** search the program performance would be degraded if

we had to search the tables **wga** and **wgb** to locate which win groups are affected by a move in order to change their state appropriately.

Data type reorder_info

The data type **reorder_info** is declared as a record of two fields: the field **wg_size** of the data type **integer** and the field **how_many** of the data type **integer**. This data type is used by the procedures **reorder_min_wg** and **reorder_moves**. A data object of the data type **reorder_info** is associated with a board position **p** and is used to store the value **TPR(C, p)** at the field **wg_size** and the value **WTPR(C, p)** at the field **how_many** of that data object, where X is in {A, B} and C is any board configuration. Recall that the functions **TPR** and **WTPR** are defined in Chapter 5.

Subprograms

Procedure initialize_free_groups

The procedure **initialize_free_groups** initializes the data objects **free_wga** and **free_wgb**. It assigns the value **TRUE** to the field **open** and the value wga(**i**).size to the field **open_positions** of the component **free_wga(i)** for each **i** in the range 1 to **wga'length** of the data object **free_wga**. Similarly it assigns the value **TRUE** to the field **open** and the value wgb(i).size to the field **open_positions** of the component **free_wgb(i)** for each **i** in the range 1 to **wgb'length** of the data object **free_wgb**.

Procedure initialize_positions

The procedure **initialize_positions** initializes the data objects: **pos_in_wga** and **pos_in_wgb**. Here we describe how it computes the value of pos_in_wga. For each position **p** in **board_range**, we locate the win groups stored at the data object **wga** that contain position p and store the identifiers of these win groups at the field **pos_in_wga(p).wg**. The computation that initializes the data object **pos_in_wgb** is similar.

Procedures update_open_wg and restore_open_wg

Procedure update_open_wg(p, m)

The procedure **update_open_wg** changes the state of all the win groups that contain position m after the occupation of the position m by player p.

The procedure **update_open_wg** takes two parameters: **p** of the data type **values** and **m** of the data type **board_range**. The parameter **p** denotes the player who moved last, **m** the last position occupied. This procedure updates the information about the **open** win groups stored in the tables: **free_wga** and **free_wgb**. The distance of each **open** win group of player **p** that contains the position **m** is reduced by one while each **open** win group of the opponent of **p** that contains position **m** becomes useless for that player. Calls to this procedure follow calls to the procedure put_value(p, m) of the package **configuration**.

Procedure restore_open_wg(p, m)

The procedure **restore_open_wg** changes the state of all the win groups that contain position **m** after the cancelation of the occupation of the position **m** by player **p**. The procedure **restore_open_wg** takes two parameters: **p** of the data type **values** and **m** of the data type **board_range**. The parameter **p** denotes the player who occupies position **m**. This procedure updates the information about the **open** win groups stored in the tables **free_wga** and **free_wgb**. The distance of each **open** win group of player **p** that contains the position **m** is increased by one while each win group of the **p**'s opponent that contains position **m**, where **m** is the only position in that win group occupied by **p**, becomes an **open** win group for that player. Calls to this procedure follow calls to the procedure **unput_value(m)** of the package **configuration**.

Function reorder_min_wg(p) returns reorder_info

The procedure **reorder_min_wg** computes the values **TPR(C, p)** and **WTPR(C, p)** for the position **p**, assigns them to the corresponding fields of a data object of the data type **reorder_info** and returns that data object to the caller. The only caller of this procedure is the procedure **reorder_moves** that calls **reorder_min_wg** anytime it needs the above values for some position **p**. We remind that X is in {A, B} and C is a board configuration.

Function alpha_beta(d, a, b) return best_rec

The function **alpha_beta** takes three parameter: **d** of the data type **integer** which denotes the current depth of the partial game tree, **a** of the data type **integer** which denotes the *alpha* bound and **b** of the data type **integer** which denotes the *beta* bound. It returns a data object of the data type **best_rec**. The field **move** of this data object is assigned the **empty** board position that if occupied it results to a board configuration which has its value stored at the field **value** of the same data object. The function **alpha_beta** is the implementation of the **alpha beta** tree search algorithm described in Chapter 5. This implementation of the **alpha beta** algorithm calls the functions: **continue_search**, **generate_moves**, **reorder_moves**, **prune_moves**, and **evaluate**. Figure 7.2 shows the function alpha_beta.

Function Continue_search(d)

The function **continue_search** takes one argument:**d** of the data type **natural** where **d** denotes the current depth of the generated partial game tree. The function **continue_search** returns a boolean value to the caller. The function determines whether the **alpha_beta** function will generate the successor board configurations of the current board configuration according to the **termination criterion** described in Chapter 5. If the game tree must be expanded one more level deeper below the current node then the function returns the value **TRUE** otherwise it returns the value **FALSE** to the caller.

Function Generate_successors returns list

The function **generate_successors** computes the **empty** positions in the current configuration. It returns a data object of the data type **list** which contains all the **empty** positions.

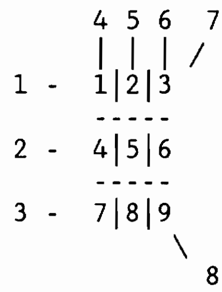
Function evaluate returns integer

The function **evaluate** is to the static evaluation function. It is called to compute the value of a terminal node (**leaf** or **tip-node**) of the generated partial game tree in a configuration C. This terminal node may or may not correspond to a terminal configuration. The function **evaluate** returns a value of data type **integer**. If the evaluated tip-node corresponds to a win (loss) configuration for the player whose turn is to move then the value **integer'last** (**integer'first**) is returned, where **integer'last** is the maximum integer value and **integer'first** is the minimum integer value. The value 0 is returned for a **draw** configuration. The value assigned to a tip-node that does not correspond to a terminal configuration depends on the concept of the **promise of a position**. The basis of the function **evaluate** has been described in Chapter 5.

Function Reorder_Moves(s) returns list

The function **reorder_moves** takes one parameter: **s** of the data type **list** and returns a data object of the data type **list**. The data object **s** stores the **empty** positions in the current configuration. The **empty** position of the current configuration are stored at the field **s.slots**.

The function **reorder_moves** copies **s** to **x**, where **x** is of the data type list, sorts the first **x.how_many** components of the array **x.slots** according to the relation **LE**, which defines a total order over the set of **empty** positions in a configuration, and returns the data object to the caller. The sorting algorithm **straight selection** is used to sort the array **x.slots**. The relation **LE** which corresponds to the reordering criterion is described in Chapter 5.



pos_in_wga

position:	1	2	3	4	5	6	7	8	9
size:	3	2	3	2	4	2	3	2	3
wg:	1 4 8	1 5	1 6 7	2 4	2 5 7 8	2 6	3 4 7	3 5	3 6 8

Figure 7.1

Data structure pos_in_wga

```
FUNCTION alpha_beta(depth        : IN integer;
                    alpha, beta : IN integer ) RETURN best_rec IS
    successors    : list;
    best, result : best_rec;
    new_move      : board_range;
  BEGIN
    best.value := alpha;
    successors := generate_moves(current_player);

    IF continue_search(depth) AND successors.how_many /= 0 THEN
      IF sc.reorder_successors THEN
        successors := reorder_moves(successors);
      END IF;

      IF sc.dead_position_test THEN
        FOR mp IN 1..successors.how_many LOOP
          IF NOT position_in_open_groups(successors.slot(mp)) THEN
                successors.how_many := mp -1 ;
                exit ;
          END IF ;
       END LOOP ;
     END IF ;

      IF sc.prune_successors AND sc.reorder_successors THEN
        successors := prune_moves(successors);
      END IF;

      FOR move IN 1 .. successors.how_many LOOP
        new_move := successors.slot(move);
        result := alpha_beta(depth + 1, -beta, -best.value) ;
        result.value := -result.value;

        IF (result.value) > best.value THEN
          best.value := result.value;
          best.move := new_move     ;
        END IF;

        IF best.value >= beta THEN
          RETURN best;
        END IF;
      END LOOP;
   ELSE
      best.value := evaluate;
   END IF;
      RETURN best;
  END alpha_beta;
```

Figure 7.2
Modified Alpha Beta Game Tree Search Algorithm

CHAPTER 8

CONCLUSION

Abstract

Chapter 8 reports the results and the conclusion for the undertaken research.

8.1 Results

This thesis produced: (a) the design of a library of software modules for the implementation of positional board game playing programs, (b) the implementation of that library and (c) the development of the programs that generated the package **board** and the sets of the **useful** win groups for the games of Hex and GO-Moku.

More specifically, we have:

(a) proved the claim that the model of positional board games presented in [Antoy 87] is the basis for building a general and reusable library of modules for the implementation of positional board game playing programs. We have proved this claim by using the library to implement two positional board games, the games of Hex and Go-Moku.

(b) identified abstract data types and operations common to all the positional board games, that are relevant to game-playing. We have also

provided formal and informal specifications of these abstract data types and operations. These specifications are the major product of this work and constitute the design of the library. These basic operations extend the model presented in [Antoy 87] which deals only with the structural characteristics of the class of the positional board games.

(c) implemented successfully the proposed abstract data type and operations of the design.

(d) identified some fundamental principles relevant to game-playing. These principles refer to board characteristics. They form the basis of the heuristic static evaluation function and the move reordering criterion of the **alpha-beta** search algorithm.

(e) proposed a representation model that captures the characteristics of any physical board, of a positional board game, that can be represented as a two dimensional array. This model allowed us to develop a common display user interface for the positional board games implemented.

(f) developed two programs, one for each of the two games implemented, that compute the sets of the **useful** win groups and generate the package **board** that corresponds to each of the games.

8.2 Conclusion

The goal of designing and implementing a general and reusable Ada library for positional board games has been accomplished. This thesis builds upon the model of positional board games presented in [Antoy 87] and extends it by adding heuristics and operations that are relevant for game playing. It also proves the claim that the model of the class of positional board games eases the development of a general library of modules that can be used as parts of programs that play the games in the class described by the model. Generally, the development of models that capture the essential structural and operational characteristics of games facilitates the development of software libraries of general and reusable modules. These modules can be used as parts of programs that play the games in the class described by the model.

CHAPTER 9

LITERATURE CITED

[Antoy 87] Antoy S., "Modeling and Isomorphisms of Positional Board Games," in IEEE Transactions on Pattern Analysis and Machine Intelligence, Vol PAMI-9, No. 5, Sept. 1987.

[Banerji 80] Banerji R.B., Artificial Intelligence: A theoretical Approach. Amsterdam, The Netherlands: North-Holland, 1980.

[Berlekamp 82] Berlekamp E. R., Conway J. H., and Guy R. K., Winning Ways for your Mathematical Plays, Volume 1: Games In General, Volume 2: Games In Particular, Academic Press 1982.

[Booch 83] Booch G., Software Enginnering with Ada. The Benjamin/Cummings Publishing Company Inc., 1983.

[Brown 89] Brown J. R. and Cunningham S., Programming the User Interface, Principles and Examples. John Wiley and Sons 1989.

[Campbel 83] Campbell M. S. and Marsland, "A Comparison of Minimax Tree Search Algorithms," in Articicial Intelligence 20 (1983), 347-367.

[Charniak 85] Charniak E. and McDermont D., Introduction to

Artificial Intelligence. Addison-Wesley, 1985.

[**Feigenbaum 81**] Feigenbaum E. A. and Barr A., The Handbook of Artificial Intelligence. Volume I. William Kaufmann, Inc., 1981.

[**Freeman 83**] Freeman P. and Wasserman A. I., Tutorial on Software Design Techniques 4th Edition, IEEE Press 1983.

[**Gardner 71**] Gardner M., Martin Gardner's Sixth Book of Mathematical Games from Scientific American. W. H. Freeman and Company, San Francisco 1971.

[**Gehani 84**] Gehani N., Ada an Advanced Introduction including Reference Manual for the Ada Programming Language. Prentice-Hall 1984.

[**Hofstadter 79**] Hofstadter D. R., Godel, Escher, Bach: An Eternal Golden Braid, Vintage Books, NY 1979.

[**Hofstadter 85**] Hofstadter D. R., Metamagical Themas: Questing for the Essence of Mind and Pattern, Basic Books Inc., NY 1985.

[**Jackson 74**] Jackson P.C. Jr., Introduction to Artificial Intelligence. Petrocelli Books, 1974.

[**Nilsson 80**] Nilsson N.J., Principles of Artificial Intelligence. Palo Alto, CA: Tioga, 1980.

[**Pearl 84**] Pearl J., Heuristics: Intlligent Search Strategies for Computer Problem Solving. Addison-Wesley Publishing Complany, 1984.

[**Rich 83**] Rich E., Artificial Intelligence. McGraw Hill, 1983.

[**Stockman 79**] Stockman G. C., "A Minimax Algorithm Better than Alpha-Beta?," in Artificial Intelligence 12 (1979), 179-196.

[**Thomas 88**] Thomas P., Robinson H., and Emms J., Abstract Data Types, Their Specification , Representation, and Use. Clarendon Press, Oxford 1988.

APPENDIX A

```
WITH board;
USE  board;

PACKAGE configuration IS
  PROCEDURE clear_board;
  PROCEDURE put_value(element : IN values; position : IN board_range);
  PROCEDURE unput_value(position : IN board_range);
  FUNCTION get_value(position : IN board_range) RETURN values;
END configuration;

PACKAGE BODY configuration IS

  TYPE board_configuration_type IS ARRAY(board_range) OF values;
  board_configuration  :  board_configuration_type  :=  (board_range  =>
empty);



  PROCEDURE clear_board IS
  BEGIN
    FOR index IN board_range LOOP
      board_configuration(index) := empty;
    END LOOP;
  END clear_board;



  FUNCTION get_value(position : IN board_range) RETURN values IS
  BEGIN
    RETURN board_configuration(position);
  END get_value;



  PROCEDURE unput_value(position : IN board_range) IS
  BEGIN
    board_configuration(position) := empty;
  END unput_value;



  PROCEDURE put_value(element : IN values; position : IN board_range) IS
  BEGIN
    board_configuration(position) := element;
  END put_value;


BEGIN
  NULL;
```

```
END configuration;
WITH configuration;
USE  configuration;

WITH board;
USE  board;

WITH move;
USE  move;

PACKAGE game IS


  TYPE options IS (quit, play, nothing, suggest, new_game, change_player,
   reorder);

  PROCEDURE play_game;

END game;
WITH text_io;
USE  text_io;

WITH integer_io;
USE  integer_io;

WITH curses;
USE  curses;

WITH user_interface;
USE  user_interface;

PACKAGE BODY game IS
  mv                : move_characteristics_type;
  request_to_stop : boolean                                    := false;
  selection         : options;
  move_counter      : integer RANGE 0 .. (number_of_positions + 1) := 0;
  move_sequence     : ARRAY(1 .. number_of_positions + 1) OF board_range;
  winner            : values;
  TYPE players IS (human, computer);
  first_player      : players                                  := human;

  PROCEDURE new_game IS
  BEGIN
    clear_board;
    move_counter := 0;
  END new_game;

  PROCEDURE update_game(move : IN board_range) IS
  BEGIN
```

```
  IF move_counter REM 2 = 0 THEN
    make_move(player_a, move);
  ELSE
    make_move(player_b, move);
  END IF;
  ---  Problem when there is a win !!!!!!!<no update>
  move_counter := move_counter + 1;
  move_sequence(move_counter) := move;
END update_game;

PROCEDURE execute_human_move IS
  move         : board_range;
  invalid_move : boolean;
  player       : values;
BEGIN
  LOOP
    invalid_move := false;
    move := get_player_move;
    IF move_counter REM 2 = 0 THEN
      player := player_a;
    ELSE
      player := player_b;
    END IF;
    IF NOT (get_value(move) = empty) THEN
      invalid_move := true;
    END IF;
    EXIT WHEN NOT invalid_move;
  END LOOP;
  update_game(move);
  update_display;
END execute_human_move;

PROCEDURE execute_machine_move IS

  move_selected : board_range;
BEGIN
  find_a_move(move_selected);
  update_game(move_selected);
  update_display;
END execute_machine_move;

PROCEDURE move IS
BEGIN
  CASE first_player IS
    WHEN computer =>
      execute_machine_move;
      execute_human_move;
    WHEN human =>
      execute_human_move;
```

```
      execute_machine_move;
  END CASE;
END move;

PROCEDURE execute(selection : IN options) IS
  smove : board_range;
BEGIN
  CASE selection IS
    WHEN new_game =>
      new_game;
      update_display;
    WHEN quit =>
      request_to_stop := true;
    WHEN reorder =>
      mv := get_move_characteristics;
      IF mv.reorder_successors THEN
        mv.reorder_successors := false;
      ELSE
        mv.reorder_successors := true;
      END IF;
      set_move_characteristics(mv);
    WHEN suggest =>
      find_a_move(smove);
      put(smove);
      new_line;
    WHEN change_player =>
      IF first_player = human THEN
        first_player := computer;
      ELSE
        first_player := human;
      END IF;
    WHEN play =>
      move;
    WHEN OTHERS =>
      NULL;
  END CASE;
EXCEPTION
  WHEN game_over =>
    execute(new_game);
END execute;

FUNCTION game_termination RETURN boolean IS
BEGIN
  IF selection = quit THEN
    RETURN true;
  ELSE
    RETURN false;
  END IF;
END game_termination;
```

```
  PROCEDURE play_game IS
  BEGIN
    init_display;
    clear_display;
    display_board;
    LOOP
      display_menu;
      selection := get_user_selection;
      execute(selection);
      EXIT WHEN request_to_stop;
    END LOOP;
  END play_game;

BEGIN
  NULL;
END game;
WITH configuration;
USE  configuration;
WITH game;
USE  game;
WITH board;
USE  board;

PACKAGE user_interface IS
  PROCEDURE init_display;
  PROCEDURE clear_display;
  PROCEDURE display_board;
  PROCEDURE display_menu;
  PROCEDURE update_display;
  PROCEDURE wait_for_keystroke;
  PROCEDURE print_winner(winner : IN values);
  PROCEDURE update_tip_nodes(tips : natural);
  FUNCTION get_player_move RETURN natural;
  FUNCTION get_user_selection RETURN options;

END user_interface;
WITH curses;
USE  curses;

WITH integer_io;
USE  integer_io;

WITH text_io;
USE  text_io;

PACKAGE BODY user_interface IS
  tip_nodes                           : natural;
  options_win, board_win, status_win : window;
```

```
depth                                     : integer;
b_ch                                      : integer;
line, column                              : board_range;
enum                                      : natural;
TYPE xy_cord IS RECORD
  x, y : integer;
END RECORD;
TYPE xy_table IS ARRAY(integer RANGE <>) OF xy_cord;
xy_display                                : xy_table(board_range);
xy_background                             : xy_table(1 .. 1000);
background                                : ARRAY(1 .. 1000) OF character;
pos_dir                                  : ARRAY(board_range, board_range) OF
 board_range;

PROCEDURE init_display IS
BEGIN
  initscr;
  options_win := newwin(18, 22, 1, 50);
  board_win := newwin(18, 45, 1, 0);
  status_win := newwin(4, 72, 19, 0);
END init_display;

PROCEDURE clear_display IS

BEGIN
  clear;
  refresh;
END clear_display;

PROCEDURE update_display IS

BEGIN
  werase(board_win);
  display_board;
END update_display;

PROCEDURE display_background IS SEPARATE;

PROCEDURE wait_for_keystroke IS
  key : character;
BEGIN
  mvwaddstr(status_win, 1, 1, "Hit any key to proceed !");
  wrefresh(status_win);
  key := mvwgetch(status_win, 2, 1);
  wclear(status_win);
  wrefresh(status_win);
END wait_for_keystroke;

PROCEDURE display_menu IS
```

```
BEGIN
  box(options_win, '*', '*');
  mvwaddstr(options_win, 1, 1, "(N)ew Game");
  mvwaddstr(options_win, 2, 1, "(S)uggest Move");
  mvwaddstr(options_win, 3, 1, "(L)evel");
  mvwaddstr(options_win, 4, 1, "(P)lay");
  mvwaddstr(options_win, 5, 1, "(C)hange_player");
  mvwaddstr(options_win, 6, 1, "(R)eorder On/Off");
  mvwaddstr(options_win, 7, 1, "(Q)uit");
  wrefresh(options_win);
END display_menu;

PROCEDURE display_board IS SEPARATE;

PROCEDURE print_winner(winner : IN values) IS
BEGIN
  wclear(status_win);
  IF winner = player_a THEN
    mvwaddstr(status_win, 2, 2, "player A is the Winner");
  ELSIF winner = player_b THEN
    mvwaddstr(status_win, 2, 2, "player B is the Winner");
  ELSE
    mvwaddstr(status_win, 2, 2, "there is  no winner");
  END IF;
  wrefresh(status_win);
END print_winner;

FUNCTION get_user_selection RETURN options IS
  selection : character;
BEGIN
  mvwaddstr(options_win, 15, 3, "Select ? ");
  wmove(options_win, 15, 11);
  wrefresh(options_win);
  selection := wgetch(options_win);
  CASE selection IS
    WHEN 'Q' | 'q' =>
      RETURN quit;
    WHEN 'P' | 'p' =>
      tip_nodes := 0;
      RETURN play;
    WHEN 'N' | 'n' =>
      RETURN new_game;
    WHEN 'S' | 's' =>
      RETURN suggest;
    WHEN 'C' | 'c' =>
      RETURN change_player;
    WHEN 'R' | 'r' =>
      RETURN reorder;
```

```
    WHEN 'L' | 'l' =>
      mvwaddstr(status_win, 2, 2, "Depth =>");
      wrefresh(status_win);
      get(depth);
    WHEN OTHERS =>
      NULL;
  END CASE;
  RETURN nothing;
END get_user_selection;

FUNCTION get_player_move RETURN natural IS
  selection : board_range;
  cp, l, c  : natural;
  cs        : string(1 .. 2);
  wch       : character;
  input_win : window;
BEGIN
  input_win := newwin(2, 20, 20, 5);
  wclear(input_win);
  wrefresh(input_win);
  mvwaddstr(input_win, 0, 1, "Enter Move !");
  cs := "11";
  cp := 1;
  LOOP
    mvwaddstr(input_win, 1, 1, "Line =>");
    wmove(input_win, 1, 8);
    wrefresh(input_win);
    wch := wgetch(input_win);
    CASE wch IS
      WHEN '1' .. '9' =>
        cs(cp) := wch;
      WHEN ' ' =>
        EXIT;
      WHEN OTHERS =>
        NULL;
    END CASE;
  END LOOP;
  l := (character'pos(cs(1)) - 48);
  cs := "11";
  cp := 1;
  wclear(input_win);
  wrefresh(input_win);
  LOOP
    box(input_win, ' ', ' ');
    mvwaddstr(input_win, 1, 1, "colm =>");
    wmove(input_win, 1, 8);
    wrefresh(input_win);
    wch := wgetch(input_win);
    CASE wch IS
```

```
        WHEN '1' .. '9' =>
          cs(cp) := wch;
        WHEN ' ' =>
          EXIT;
        WHEN OTHERS =>
          NULL;
      END CASE;
    END LOOP;
    c := character'pos(cs(1)) - 48;
    wclear(input_win);
    wrefresh(input_win);
    delwin(input_win);
    RETURN pos_dir(1, c);
  END get_player_move;
  PROCEDURE init_display_coordinates IS SEPARATE;

  PROCEDURE update_tip_nodes(tips : natural) IS
  BEGIN
    wclear(status_win);
    wmove(status_win, 1, 40);
    wrefresh(status_win);
    put(tips, 5);
  END update_tip_nodes;
BEGIN
  tip_nodes := 0;
  init_display_coordinates;
END user_interface;
SEPARATE(user_interface)

PROCEDURE display_board IS
  player : values;
  k      : natural;
BEGIN
  box(board_win, '*', '*');
  wmove(board_win, xy_display(1).x - 2, xy_display(1).y - 1);
  wrefresh(board_win);
  FOR lines IN 1 .. line LOOP
    put(lines, 2);
  END LOOP;
  k := 1;
  FOR col IN 1 .. column LOOP
    wmove(board_win, xy_display(k).x, xy_display(k).y - 3);
    wrefresh(board_win);
    k := k + line;
    put(col, 2);
  END LOOP;
  FOR index IN board_range LOOP
    player := get_value(index);
    IF player = player_a THEN
```

```
      mvwaddstr(board_win, xy_display(index).x, xy_display(index).y, "A");
    ELSIF player = player_b THEN
      mvwaddstr(board_win, xy_display(index).x, xy_display(index).y, "B");
    ELSE
      mvwaddstr(board_win, xy_display(index).x, xy_display(index).y, "X");
    END IF;
  END LOOP;
  display_background;
  wrefresh(board_win);
END display_board;


SEPARATE(user_interface)

PROCEDURE display_background IS
  backchar : string(1 .. 1);
BEGIN
  FOR bc IN 1 .. b_ch LOOP
    backchar(1) := background(bc);
    mvwaddstr(board_win,    xy_background(bc).x,    xy_background(bc).y,
backchar);
  END LOOP;
END display_background;
SEPARATE(move)

FUNCTION evaluate RETURN weight IS
  holes                                                    : natural;
  suma, sumb                                               : weight;
  value_a                                                  :
   ARRAY(0 .. max_holes_in_a) OF natural := (0 .. max_holes_in_a => 0);
  value_b                                                  :
   ARRAY(0 .. max_holes_in_b) OF natural := (0 .. max_holes_in_b => 0);
  open_wg_a_h0, open_wg_b_h0, open_wg_a_h1, open_wg_b_h1 : integer;
  promise                                                         :
ARRAY(board_range,
   player_a .. player_b) OF natural;
  FUNCTION find_owg_min_holes(p      : board_range;
                              player : values) RETURN natural IS
    min_holes : natural;
    wid       : natural;
  BEGIN
    IF player = player_a THEN
      min_holes := max_holes_in_a;
      FOR w IN 1 .. pos_in_wga(p).size LOOP
        wid := pos_in_wga(p).wg(w);
        IF free_wga(wid).open AND free_wga(wid).open_positions <
         min_holes THEN
          min_holes := free_wga(wid).open_positions;
        END IF;
```

```
      END LOOP;
    ELSIF player = player_b THEN
      min_holes := max_holes_in_b;
      FOR w IN 1 .. pos_in_wgb(p).size LOOP
        wid := pos_in_wgb(p).wg(w);
        IF free_wgb(wid).open AND free_wgb(wid).open_positions <
         min_holes THEN
          min_holes := free_wgb(wid).open_positions;
        END IF;
      END LOOP;
    ELSE
      NULL;
    END IF;
    RETURN min_holes;
  END find_owg_min_holes;

BEGIN
  suma := weight(0);
  sumb := weight(0);
  open_wg_a_h0 := open_wg(0, player_a);
  open_wg_b_h0 := open_wg(0, player_b);
  IF current_player = player_a THEN
    IF open_wg_a_h0 > 0 THEN
      RETURN weight'last;
    END IF;
    IF open_wg_b_h0 > 0 THEN
      RETURN -weight'last;
    END IF;
  ELSE
    IF open_wg_b_h0 > 0 THEN
      RETURN weight'last;
    END IF;
    IF open_wg_a_h0 > 0 THEN
      RETURN -weight'last;
    END IF;
  END IF;
  FOR p IN 1 .. number_of_positions LOOP
    IF get_value(p) = empty THEN
      holes := find_owg_min_holes(p, player_a);
      value_a(holes) := value_a(holes) + 1;
      holes := find_owg_min_holes(p, player_b);
      value_b(holes) := value_b(holes) + 1;
    ELSE
      NULL;
    END IF;
  END LOOP;
  FOR h IN 1 .. max_holes_in_a LOOP
    IF value_a(h) > 0 THEN
      suma := suma + weight(value_a(h) + (2**(max_holes_in_a - h) -1) *
```

```
        number_of_positions);
      END IF;
    END LOOP;
    FOR h IN 1 .. max_holes_in_b LOOP
      IF value_b(h) > 0 THEN
        sumb := sumb + weight(value_b(h) + (2**(max_holes_in_b - h) -1) *
         number_of_positions);
      END IF;
    END LOOP;
    IF current_player = player_a THEN
      RETURN suma - sumb;
    ELSE
      RETURN sumb - suma;
    END IF;
  END evaluate;
  WITH board;

  USE  board;

  WITH configuration;
  USE  configuration;
  PACKAGE move IS

    winner_a, winner_b, draw, game_over : EXCEPTION;

    TYPE move_characteristics_type IS RECORD
      reorder_successors : boolean;
      prune_successors   : boolean;
      alive_nodes_test   : boolean;
      dead_position_test : boolean;
      search_depth_limit : natural;
      trace_search       : boolean;
      trace_file         : string(1 .. 10);
    END RECORD;

    FUNCTION get_move_characteristics RETURN move_characteristics_type;
    PROCEDURE    set_move_characteristics(new_characteristics    :    IN
  move_characteristics_type);
    PROCEDURE find_a_move(suggested_move : OUT board_range);
    PROCEDURE make_move(player : IN values; move : IN board_range);

  END move;
  WITH user_interface;
  USE  user_interface;

  WITH curses;
  USE  curses;
  WITH integer_io;
  USE  integer_io;
```

```
WITH text_io;
USE  text_io;

PACKAGE BODY move IS
  bad_call_parameters : EXCEPTION;

  TYPE board_status_rec IS RECORD
    winner          : values;
    terminated      : boolean;
    player_to_move  : values;
  END RECORD;
  TYPE reorder_info IS RECORD
    wg_size         : integer;
    how_many        : integer;
  END RECORD;

  TYPE   counter            IS ARRAY(integer RANGE <>) OF natural;
  TYPE open_wg_type IS RECORD
    open            : boolean;
    open_positions  : integer;
  END RECORD;

  wg_size_a           : CONSTANT integer          := wga'length;
  wg_size_b           : CONSTANT integer          := wgb'length;

  biggest_wg_size     : integer;

  SUBTYPE max_range         IS integer RANGE 1 .. biggest_wg_size;

  current_player      : values;
  bc                  : move_characteristics_type :=
   (reorder_successors => true, prune_successors => true,
   alive_nodes_test => true, search_depth_limit => 3,
   dead_position_test => true, trace_search => false,
   trace_file => "xxxx.trace");


  SUBTYPE list_range        IS natural RANGE 0 .. 100;
  TYPE    slot_type         IS ARRAY(list_range) OF board_range;
  TYPE list IS RECORD
    how_many        : list_range;
    slot            : slot_type;
  END RECORD;

  winner              : values                    := empty;

  max_holes_in_a_wg   : integer;
  many_open_positions : CONSTANT integer          := 10000;
```

```
free_wga            : ARRAY(1 .. wg_size_a) OF open_wg_type;
free_wgb            : ARRAY(1 .. wg_size_b) OF open_wg_type;
pos_in_wga          : wg_table(1 .. number_of_positions);
pos_in_wgb          : wg_table(1 .. number_of_positions);


SUBTYPE weight            IS short_float;
TYPE    weight_table_type IS ARRAY(integer RANGE <>) OF weight;

max_holes_in_a      : natural;
max_holes_in_b      : natural;


weights_for_a       : weight_table_type(board_range);
weights_for_b       : weight_table_type(board_range);

PROCEDURE change_current_player IS
BEGIN
  IF current_player = player_a THEN
    current_player := player_b;
  ELSE
    current_player := player_a;
  END IF;
END change_current_player;


FUNCTION get_move_characteristics RETURN move_characteristics_type IS
BEGIN
  RETURN bc;
END get_move_characteristics;

PROCEDURE    set_move_characteristics(new_characteristics    :    IN
move_characteristics_type) IS
BEGIN
  bc := new_characteristics;
END set_move_characteristics;


PROCEDURE initialize_free_groups IS SEPARATE;

FUNCTION check_if_open(player : values;
                       wg_id  : integer) RETURN boolean IS SEPARATE;
FUNCTION position_in_open_groups(position : IN board_range)
                                 RETURN boolean IS SEPARATE;
FUNCTION reorder_min_wg(position : IN board_range)
                        RETURN reorder_info IS SEPARATE;
FUNCTION who_is_winner RETURN values IS SEPARATE;
FUNCTION wg_combinations(player : values) RETURN counter IS SEPARATE;
```

```
FUNCTION free_wg_of_size(critical_size : IN integer)
                         RETURN boolean IS SEPARATE;
FUNCTION open_wg(open_pos : IN integer;
                 player   : IN values) RETURN integer IS SEPARATE;
FUNCTION someone_won RETURN values IS SEPARATE;
PROCEDURE update_open_wg(player : IN values;
                         move   : IN board_range) IS SEPARATE;
PROCEDURE restore_open_wg(player : IN values;
                          move   : IN board_range) IS SEPARATE;

PROCEDURE print_wg IS SEPARATE;


PROCEDURE initialize_positions IS SEPARATE;



FUNCTION test_termination RETURN boolean IS SEPARATE;

FUNCTION board_status RETURN board_status_rec IS
  back : board_status_rec;
BEGIN
  back.winner := someone_won;
  back.terminated := test_termination;
  back.player_to_move := current_player;
  CASE back.winner IS
    WHEN player_a =>
      RAISE winner_a;
    WHEN player_b =>
      RAISE winner_b;
    WHEN empty =>
      IF back.terminated THEN
        RAISE draw;
      END IF;
    WHEN OTHERS =>
      NULL;
  END CASE;
  RETURN back;
END board_status;


FUNCTION calculate_weights(player : values)
                           RETURN weight_table_type IS SEPARATE;


FUNCTION evaluate RETURN weight IS SEPARATE;

FUNCTION verify_rules(player   : IN values;
                      position : IN board_range) RETURN boolean IS
```

```
BEGIN
  RETURN true;
END verify_rules;

FUNCTION valid(player : IN values;
               move   : IN board_range) RETURN boolean IS SEPARATE;

PROCEDURE unput_element(position : IN board_range) IS
BEGIN
  unput_value(position);
  change_current_player;
  restore_open_wg(current_player, position);
END unput_element;



PROCEDURE put_element(element : IN values; position : IN board_range) IS
BEGIN
  put_value(element, position);
  update_open_wg(element, position);
  change_current_player;
END put_element;

PROCEDURE make_move(player : IN values; move : IN board_range) IS
  stat : board_status_rec;
BEGIN
  put_element(player, move);
  stat := board_status;
EXCEPTION
  WHEN winner_a =>
    current_player := player_a;
    display_board;
    print_winner(player_a);
    wait_for_keystroke;
    initialize_free_groups;
    clear_board;
    RAISE game_over;
  WHEN winner_b =>
    current_player := player_a;
    display_board;
    print_winner(player_b);
    wait_for_keystroke;
    initialize_free_groups;
    clear_board;
    RAISE game_over;
  WHEN draw =>
    current_player := player_a;
    display_board;
```

```
      print_winner(empty);
      wait_for_keystroke;
      initialize_free_groups;
      clear_board;
      RAISE game_over;
    WHEN OTHERS =>
      NULL;
  END make_move;

  PACKAGE search IS

    FUNCTION search_game_tree RETURN board_range;

  END search;

  PACKAGE BODY search IS SEPARATE;
  USE  search;


  PROCEDURE find_a_move(suggested_move : OUT board_range) IS
    stat : board_status_rec;
  BEGIN
    stat := board_status;
    suggested_move := search_game_tree;
  END find_a_move;

BEGIN
  IF wg_size_a > wg_size_b THEN
    biggest_wg_size := wg_size_a;
  ELSE
    biggest_wg_size := wg_size_b;
  END IF;
  current_player := player_a;
  max_holes_in_a := 0;
  max_holes_in_b := 0;
  FOR index IN 1 .. wg_size_a LOOP
    IF wga(index) /= NULL THEN
      IF wga(index).size > max_holes_in_a THEN
        max_holes_in_a := wga(index).size;
      END IF;
    END IF;
  END LOOP;
  FOR index IN wgb'RANGE LOOP
    IF wgb(index).size > max_holes_in_b THEN
      max_holes_in_b := wgb(index).size;
    END IF;
  END LOOP;
  max_holes_in_a_wg := max_holes_in_b;
  IF max_holes_in_a > max_holes_in_b THEN
```

```
    max_holes_in_a_wg := max_holes_in_a;
  END IF;
  initialize_positions;
  initialize_free_groups;
  weights_for_a(1 .. max_holes_in_a) := calculate_weights(player_a);
  weights_for_b(1 .. max_holes_in_b) := calculate_weights(player_b);
END move;
SEPARATE(move)


PACKAGE BODY search IS

  SUBTYPE depth_type IS integer RANGE 1 .. 100;
  alive_size    : CONSTANT natural := 1;
  nodes_visited : integer          := 0;
  tip_nodes     : integer          := 0;
  big_num       : CONSTANT weight  := weight'last;

  TYPE best_rec IS RECORD
    value : weight;
    move  : board_range;
  END RECORD;


  FUNCTION generate_moves(player_to_move : IN values) RETURN list IS
    free_list : list;
  BEGIN
    free_list.how_many := 0;
    FOR index IN board_range LOOP
      IF valid(player_to_move, index) THEN
        free_list.how_many := free_list.how_many + 1;
        free_list.slot(free_list.how_many) := index;
      END IF;
    END LOOP;
    RETURN free_list;
  END generate_moves;

  FUNCTION reorder_moves(moves_list : IN list) RETURN list IS
    reorder_nodes_info : ARRAY(1 .. moves_list.how_many) OF reorder_info;
    temp_info          : reorder_info;
    temp_pos           : board_range;
    reordered_list     : list := moves_list;
    index_2, pivot     : integer;
  BEGIN
    FOR index IN 1 .. moves_list.how_many LOOP
      reorder_nodes_info(index):=reorder_min_wg(moves_list.slot(index));
    END LOOP;
    FOR index IN 1 .. moves_list.how_many - 1 LOOP
      index_2 := index;
```

```
    pivot := index;
    FOR index_3 IN index + 1 .. moves_list.how_many LOOP
      IF reorder_nodes_info(index_3).wg_size <
       reorder_nodes_info(pivot).wg_size OR
       (reorder_nodes_info(index_3).wg_size =
       reorder_nodes_info(pivot).wg_size AND
       reorder_nodes_info(index_3).how_many >
       reorder_nodes_info(pivot).how_many) THEN
        index_2 := index_3;
        pivot := index_3;
      END IF;
      temp_info := reorder_nodes_info(pivot);
      temp_pos := reordered_list.slot(pivot);
      reordered_list.slot(index_2) := reordered_list.slot(index);
      reorder_nodes_info(index_2) := reorder_nodes_info(index);
      reordered_list.slot(index) := temp_pos;
      reorder_nodes_info(index) := temp_info;
    END LOOP;
  END LOOP;
  --for index in 1..moves_list.how_many loop
  --put(moves_list.slot(index)); put(reordered_list.slot(index));
  --new_line ;
  --put(reorder_nodes_info(index).wg_size);
  --put(reorder_nodes_info(index).how_many) ; new_line ;
  --end loop ;
  RETURN reordered_list;
END reorder_moves;


FUNCTION prune_moves(moves_list : IN list) RETURN list IS
  pruned_list : list;
BEGIN
  IF moves_list.how_many > 20 THEN
    pruned_list.how_many := 20;
    pruned_list.slot(1 .. 20) := moves_list.slot(1 .. 20);
  ELSE
    pruned_list := moves_list;
  END IF;
  RETURN pruned_list;
END prune_moves;

-- this function continues the search on level deeper if there are
-- successors since there is a reason to improve the game! move.


FUNCTION deep_enough(depth : integer) RETURN boolean IS
BEGIN
  IF (depth < bc.search_depth_limit) THEN
    RETURN false;
```

```
  ELSIF bc.alive_nodes_test THEN
    RETURN (NOT free_wg_of_size(alive_size));
  ELSE
    RETURN true;
  END IF;
END deep_enough;

-- this function continues the search if a maximum level has not
-- been reached and there is no winning configuration.

FUNCTION continue_search(depth : integer) RETURN boolean IS
BEGIN
  IF (NOT deep_enough(depth)) AND (someone_won = empty) THEN
    RETURN true;
  ELSE
    RETURN false;
  END IF;
END continue_search;


FUNCTION alpha_beta(depth       : IN integer;
                    alpha, beta : IN weight) RETURN best_rec IS
  successors   : list;
  best, result : best_rec;
  next_player  : values;
  new_move     : board_range;
BEGIN
  best.value := alpha;
  successors := generate_moves(current_player);

  IF continue_search(depth) AND successors.how_many /= 0 THEN
    IF bc.reorder_successors THEN
      successors := reorder_moves(successors);
    END IF;
    IF bc.dead_position_test THEN
      FOR mp IN 1 .. successors.how_many LOOP
        IF NOT position_in_open_groups(successors.slot(mp)) THEN
          successors.how_many := mp - 1;
          EXIT;
        END IF;
      END LOOP;
    END IF;

    IF bc.prune_successors AND bc.reorder_successors THEN
      successors := prune_moves(successors);
    END IF;

    FOR move IN 1 .. successors.how_many LOOP
      new_move := successors.slot(move);
```

```
      put_element(current_player, new_move);
      nodes_visited := nodes_visited + 1;
      result := alpha_beta(depth + 1, -beta, -best.value);
      result.value := -result.value;
      unput_element(new_move);
      IF (result.value) > best.value THEN
        best.value := result.value;
        best.move := new_move;
      END IF;
      IF best.value >= beta THEN
        RETURN best;
      END IF;
    END LOOP;
    RETURN best;
  ELSE
    tip_nodes := tip_nodes + 1;
    best.value := evaluate;
  END IF;
  RETURN best;
END alpha_beta;


FUNCTION search_game_tree RETURN board_range IS
  proposed_move : board_range;
  move_rec      : best_rec;
  candidates    : list;
BEGIN
  nodes_visited := 0;
  tip_nodes := 0;
                                                --    candidates :=
                                                         --
generate_moves(cur-
                                                --    rent_player);
                                                --    candidates :=
                                                         --
reorder_moves(cand-
                                                --    idates);
  proposed_move := candidates.slot(1);
  move_rec := alpha_beta(0, -big_num, big_num);
  update_tip_nodes(tip_nodes);
  RETURN move_rec.move;
END search_game_tree;
BEGIN
  NULL;
END search;
SEPARATE(move)
PROCEDURE initialize_positions IS
  how_many : natural;
  groups   : ARRAY(1 .. biggest_wg_size) OF natural;
```

```
BEGIN
  FOR p_index IN board_range LOOP
    how_many := 0;
    FOR w_index IN wga'RANGE LOOP
      FOR index IN 1 .. wga(w_index).size LOOP
        IF p_index = wga(w_index).wg(index) THEN
          how_many := how_many + 1;
          groups(how_many) := w_index;
        END IF;
      END LOOP;
    END LOOP;
    pos_in_wga(p_index) := NEW wg_type(size => how_many);
    FOR index IN 1 .. how_many LOOP
      pos_in_wga(p_index).wg(index) := groups(index);
    END LOOP;
  END LOOP;
  FOR p_index IN board_range LOOP
    how_many := 0;
    FOR w_index IN wgb'RANGE LOOP
      FOR index IN 1 .. wgb(w_index).size LOOP
        IF p_index = wgb(w_index).wg(index) THEN
          how_many := how_many + 1;
          groups(how_many) := w_index;
        END IF;
      END LOOP;
    END LOOP;
    pos_in_wgb(p_index) := NEW wg_type(size => how_many);
    FOR index IN 1 .. how_many LOOP
      pos_in_wgb(p_index).wg(index) := groups(index);
    END LOOP;
  END LOOP;
  --for index in board_range loop
  --   for index2 in 1..pos_in_wga(index).size loop
  --       put(pos_in_wga(index).wg(index2)) ;
  --   end loop ;
  --new_line ;
  --end loop ;
  --for index in board_range loop
  --   for index2 in 1..pos_in_wgb(index).size loop
  --put(pos_in_wgb(index).wg(index2)) ;
  -- end loop ;
  --new_line ;
  --end loop ;
END initialize_positions;

SEPARATE(move)
PROCEDURE initialize_free_groups IS
BEGIN
  FOR index IN wga'RANGE LOOP
```

```
    free_wga(index).open := true;
    free_wga(index).open_positions := wga(index).size;
  END LOOP;
  FOR index IN wgb'RANGE LOOP
    free_wgb(index).open := true;
    free_wgb(index).open_positions := wgb(index).size;
  END LOOP;
END initialize_free_groups;

SEPARATE(move)

FUNCTION reorder_min_wg(position : IN board_range) RETURN reorder_info IS
  size, wg_id    : integer;
  min_wg         : reorder_info;
  free_positions : integer;
BEGIN
  min_wg.wg_size := many_open_positions;
  min_wg.how_many := 0;
  --    if current_player = player_a then
  size := pos_in_wga(position).size;
  FOR index IN 1 .. size LOOP
    wg_id := pos_in_wga(position).wg(index);
    IF free_wga(wg_id).open THEN
      free_positions := free_wga(wg_id).open_positions;
      IF free_positions < min_wg.wg_size THEN
        min_wg.wg_size := free_positions;
        min_wg.how_many := 1;
      ELSIF free_positions = min_wg.wg_size THEN
        min_wg.how_many := min_wg.how_many + 1;
      ELSE
        NULL;
      END IF;
    END IF;
  END LOOP;
  --else
  size := pos_in_wgb(position).size;
  FOR index IN 1 .. size LOOP
    wg_id := pos_in_wgb(position).wg(index);
    IF free_wgb(wg_id).open THEN
      free_positions := free_wgb(wg_id).open_positions;
      IF free_positions < min_wg.wg_size THEN
        min_wg.wg_size := free_positions;
        min_wg.how_many := 1;
      ELSIF free_positions = min_wg.wg_size THEN
        min_wg.how_many := min_wg.how_many + 1;
      ELSE
        NULL;
      END IF;
    END IF;
```

```
  END LOOP;
  --end if ;
  RETURN min_wg;
END reorder_min_wg;

SEPARATE(move)
FUNCTION who_is_winner RETURN values IS
BEGIN
  RETURN winner;
END who_is_winner;

SEPARATE(move)
FUNCTION someone_won RETURN values IS
  is_there_a_winner : boolean;
BEGIN
  is_there_a_winner := false;
  FOR index IN 1 .. wg_size_a LOOP
    IF free_wga(index).open = true AND free_wga(index).open_positions = 0
THEN
      winner := player_a;
      is_there_a_winner := true;
      RETURN player_a;
    END IF;
  END LOOP;
  FOR index IN 1 .. wg_size_b LOOP
    IF free_wgb(index).open = true AND free_wgb(index).open_positions = 0
THEN
      winner := player_b;
      RETURN player_b;
    END IF;
  END LOOP;
  RETURN empty;
END someone_won;

SEPARATE(move)
FUNCTION free_wg_of_size(critical_size : IN integer) RETURN boolean IS
BEGIN
  IF current_player = player_a THEN
    FOR free_index IN 1 .. wg_size_a LOOP
      IF free_wga(free_index).open = true AND
       free_wga(free_index).open_positions <= critical_size THEN
      --put("player A => group # is ->") ; put(free_index) ; new_line ;
        RETURN true;
      END IF;
    END LOOP;
  ELSE
    FOR free_index IN 1 .. wg_size_b LOOP
      IF free_wgb(free_index).open = true AND
       free_wgb(free_index).open_positions <= critical_size THEN
```

```
      --+++put("player b => group # is ->") ; put(free_index) ; new_line
;
        RETURN true;
      END IF;
    END LOOP;
  END IF;
  RETURN false;
END free_wg_of_size;

SEPARATE(move)
FUNCTION open_wg(open_pos : IN integer; player : IN values) RETURN integer
IS
  count_open : integer;
BEGIN
  count_open := 0;
  IF player = player_a THEN
    FOR index IN 1 .. wg_size_a LOOP
      IF free_wga(index).open = true AND free_wga(index).open_positions =
       open_pos THEN
        count_open := count_open + 1;
      END IF;
    END LOOP;
    RETURN count_open;
  ELSE
    count_open := 0;
    FOR index IN 1 .. wg_size_b LOOP
      IF free_wgb(index).open = true AND free_wgb(index).open_positions =
       open_pos THEN
        count_open := count_open + 1;
      END IF;
    END LOOP;
    RETURN count_open;
  END IF;
END open_wg;

SEPARATE(move)
FUNCTION position_in_open_groups(position : IN board_range) RETURN boolean
IS
  wg_id : integer;
  size  : integer;
BEGIN
  size := pos_in_wga(position).size;
  FOR index IN 1 .. size LOOP
    wg_id := pos_in_wga(position).wg(index);
    IF free_wga(wg_id).open THEN
      RETURN true;
    END IF;
  END LOOP;
  size := pos_in_wgb(position).size;
```

```
  FOR index IN 1 .. size LOOP
    wg_id := pos_in_wgb(position).wg(index);
    IF free_wgb(wg_id).open THEN
      RETURN true;
    END IF;
  END LOOP;
  RETURN false;
END position_in_open_groups;

SEPARATE(move)
FUNCTION check_if_open(player : values; wg_id : integer) RETURN boolean IS
  pos_value : values;
BEGIN
  IF player = player_a THEN
    FOR index IN 1 .. wga(wg_id).size LOOP
      pos_value := get_value(wga(wg_id).wg(index));
      IF pos_value = player_b THEN
        RETURN false;
      END IF;
    END LOOP;
  ELSE
    FOR index IN 1 .. wgb(wg_id).size LOOP
      pos_value := get_value(wgb(wg_id).wg(index));
      IF pos_value = player_a THEN
        RETURN false;
      END IF;
    END LOOP;
  END IF;
  RETURN true;
END check_if_open;

SEPARATE(move)
FUNCTION wg_combinations(player : values) RETURN counter IS
  SUBTYPE win_range IS integer RANGE 0 .. max_holes_in_a_wg;
  SUBTYPE max_range IS integer RANGE 1 .. biggest_wg_size;
  win, freesize, possible_wgs : counter(win_range) := (win_range => 0);
BEGIN
  IF player = player_a THEN
    FOR index IN win_range LOOP
      FOR wg_index IN wga'RANGE LOOP
        IF wga(wg_index).size >= index THEN
          possible_wgs(index) := possible_wgs(index) + 1;
        END IF;
      END LOOP;
    END LOOP;
    RETURN possible_wgs(1 .. max_holes_in_a);
  ELSE
    FOR index IN win_range LOOP
      FOR wg_index IN wgb'RANGE LOOP
```

```
        IF wgb(wg_index).size >= index THEN
          possible_wgs(index) := possible_wgs(index) + 1;
        END IF;
      END LOOP;
    END LOOP;
    RETURN possible_wgs(1 .. max_holes_in_b);
  END IF;
END wg_combinations;

SEPARATE(move)
PROCEDURE update_open_wg(player : IN values; move : IN board_range) IS
  wg_id : integer RANGE 1 .. 500;    --  potential problem
BEGIN
  IF player = player_a THEN
    FOR index IN 1 .. pos_in_wga(move).size LOOP
      wg_id := pos_in_wga(move).wg(index);
      free_wga(wg_id).open_positions := free_wga(wg_id).open_positions -
1;
    END LOOP;
    FOR index IN 1 .. pos_in_wgb(move).size LOOP
      wg_id := pos_in_wgb(move).wg(index);
      free_wgb(wg_id).open_positions := free_wgb(wg_id).open_positions -
1;
      free_wgb(wg_id).open := false;
    END LOOP;
  ELSE
    FOR index IN 1 .. pos_in_wga(move).size LOOP
      wg_id := pos_in_wga(move).wg(index);
      free_wga(wg_id).open_positions := free_wga(wg_id).open_positions -
1;
      free_wga(wg_id).open := false;
    END LOOP;
    FOR index IN 1 .. pos_in_wgb(move).size LOOP
      wg_id := pos_in_wgb(move).wg(index);
      free_wgb(wg_id).open_positions := free_wgb(wg_id).open_positions -
1;
    END LOOP;
  END IF;
END update_open_wg;

SEPARATE(move)
PROCEDURE restore_open_wg(player : IN values; move : IN board_range) IS
BEGIN

  IF player = player_a THEN
    FOR index IN 1 .. pos_in_wga(move).size LOOP
      free_wga(pos_in_wga(move).wg(index)).open_positions :=
       free_wga(pos_in_wga(move).wg(index)).open_positions + 1;
    END LOOP;
```

```
    FOR index IN 1 .. pos_in_wgb(move).size LOOP
      free_wgb(pos_in_wgb(move).wg(index)).open_positions :=
       free_wgb(pos_in_wgb(move).wg(index)).open_positions + 1;
      free_wgb(pos_in_wgb(move).wg(index)).open:=check_if_open(player_b,
       pos_in_wgb(move).wg(index));
    END LOOP;
  ELSE
    FOR index IN 1 .. pos_in_wga(move).size LOOP
      free_wga(pos_in_wga(move).wg(index)).open_positions :=
       free_wga(pos_in_wga(move).wg(index)).open_positions + 1;
      free_wga(pos_in_wga(move).wg(index)).open:=check_if_open(player_a,
       pos_in_wga(move).wg(index));
    END LOOP;
    FOR index IN 1 .. pos_in_wgb(move).size LOOP
      free_wgb(pos_in_wgb(move).wg(index)).open_positions :=
       free_wgb(pos_in_wgb(move).wg(index)).open_positions + 1;
    END LOOP;
  END IF;
END restore_open_wg;

SEPARATE(move)
PROCEDURE print_wg IS
BEGIN
  new_line;
  put("FREE WG");
  new_line;
  FOR index IN 1 .. wg_size_a LOOP
    put(free_wga(index).open_positions);
    IF free_wga(index).open THEN
      put("TRUE");
    ELSE
      put("F");
    END IF;
  END LOOP;
  new_line;
  FOR index IN 1 .. wg_size_b LOOP
    put(free_wgb(index).open_positions);
    IF free_wgb(index).open THEN
      put("TRUE");
    ELSE
      put("F");
    END IF;
  END LOOP;
  new_line;
END print_wg;
SEPARATE(move)

FUNCTION test_termination RETURN boolean IS
  free_positions : boolean;
```

```
BEGIN
  free_positions := false;
  FOR index IN board_range LOOP
    IF get_value(index) = empty THEN
      free_positions := true;
      EXIT;
    END IF;
  END LOOP;
  IF NOT free_positions OR (someone_won /= empty) THEN
    RETURN true;
  ELSE
    RETURN false;
  END IF;
END test_termination;

SEPARATE(move)
FUNCTION valid(player : IN values; move : IN board_range) RETURN boolean
IS
BEGIN
  IF (get_value(move) = empty) AND verify_rules(player, move) THEN
    RETURN true;
  ELSE
    RETURN false;
  END IF;
END valid;
WITH my_board;
USE  my_board;

FUNCTION verify_rules(player   : IN my_board.values;
                      position : IN board_range) RETURN boolean IS

BEGIN
  RETURN true;
END verify_rules;
WITH game;
USE  game;
PROCEDURE x_game IS
BEGIN
  play_game;
END x_game;
WITH integer_io;
USE  integer_io;

WITH text_io;
USE  text_io;


PROCEDURE hex IS
  TYPE    player_type IS (a, b);
```

```
wg_limit_size          : integer;
TYPE    wg_array    IS ARRAY(integer RANGE <>) OF integer;
SUBTYPE max_range   IS integer RANGE 1 .. 1000;
wg_size                : ARRAY(player_type) OF integer := (0, 0);
max_wg                 : ARRAY(player_type) OF integer := (0, 0);
line_size              : CONSTANT integer := 5;
big                    : integer;
board_size             : CONSTANT integer := line_size * line_size;
SUBTYPE board_range IS integer RANGE 1 .. board_size;
SUBTYPE line_range  IS integer RANGE 1 .. line_size;
TYPE pos_type IS RECORD
  size : integer;
  wg   : wg_array(1 .. 100);
END RECORD;

TYPE wg_t IS RECORD
  size : integer;
  wg   : wg_array(1 .. board_size / 2 + 1);
END RECORD;
null_pos               : pos_type         := (size => 0,
 wg => (1 .. 100 => 0));
null_wg                : wg_t             := (size => 0,
 wg => (1 .. board_size / 2 + 1 => 0));
pos                    : ARRAY(player_type, board_range) OF pos_type :=
 (player_type => (board_range => null_pos));
wg                     : ARRAY(player_type, max_range) OF wg_t :=
 (player_type => (max_range => null_wg));
TYPE    start_up    IS ARRAY(player_type, line_range) OF board_range;
SUBTYPE list_range  IS integer RANGE 0 .. 6;
TYPE    slots_type  IS ARRAY(1 .. 6) OF board_range;
TYPE list IS RECORD
  size  : list_range;
  slots : slots_type;
END RECORD;
start, ending          : start_up;
start_b, end_a, end_b : integer;    --to be changed to reflect
depth                  : board_range      := 1;
TYPE    path_type   IS ARRAY(board_range) OF integer;
path                   : path_type;
occupied               : ARRAY(board_range) OF boolean :=
 (board_range => false);
modifiers                   : ARRAY(1 .. 6) OF integer := (-1, + 1,
-(line_size -
 1), (line_size - 1), -line_size, line_size);

FUNCTION check_end_of_path(player   : player_type;
                           position : board_range) RETURN boolean IS
BEGIN
  FOR index IN line_range LOOP
```

```
    IF ending(player, index) = position THEN
      RETURN true;
    END IF;
  END LOOP;
  RETURN false;
END check_end_of_path;

FUNCTION line(a : IN board_range) RETURN integer IS
BEGIN
  IF (a REM line_size) /= 0 THEN
    RETURN (a / line_size + 1);
  ELSE
    RETURN (a / line_size);
  END IF;
END line;

FUNCTION is_connected(first, second : IN board_range) RETURN boolean IS
  candidate : integer;
BEGIN
  FOR index IN 1 .. 6 LOOP
    candidate := first + modifiers(index);
    IF candidate >= 1 AND candidate <= board_size THEN
      CASE modifiers(index) IS
        WHEN -1 | 1 =>
          IF line(candidate) = line(first) THEN
            IF candidate = second THEN
              RETURN true;
            END IF;
          END IF;
        WHEN -line_size | line_size | -line_size + 1 | line_size - 1 =>
          IF line(candidate) /= line(first) THEN
            IF candidate = second THEN
              RETURN true;
            END IF;
          END IF;
        WHEN OTHERS =>
          NULL;
      END CASE;
    END IF;
  END LOOP;
  RETURN false;
END is_connected;

FUNCTION surrounding_positions(position : board_range) RETURN list IS
  neighbors : list := (size => 0, slots => (1 .. 6 => 1));
  size      : integer RANGE 0 .. 6;
  candidate : integer;
BEGIN
  neighbors.size := 0;
```

```
  --put("successors of"); put(position ) ; ne  _line ;
  FOR index IN 1 .. 6 LOOP
    candidate := position + modifiers(index);
    --put("cand =>"); put(candidate) ; new_line ;
    IF candidate >= 1 AND candidate <= board_size THEN
      CASE modifiers(index) IS
        WHEN -1 | 1 =>
          IF line(candidate) = line(position) THEN
          --         put(candidate) ; put("SELECTED") ; new_line ;
            neightbors.size := neightbors.size + 1;
            size := neightbors.size;
            neightbors.slots(size) := candidate;
          END IF;
        WHEN -line_size | line_size | -line_size + 1 | line_size - 1 =>
          IF line(candidate) /= line(position) THEN
          --         put(candidate) ; put("SELECTED") ; new_line ;
            neightbors.size := neightbors.size + 1;
            size := neightbors.size;

            neightbors.slots(size) := candidate;
          END IF;
        WHEN OTHERS =>
          NULL;
      END CASE;
    END IF;
  END LOOP;
  --   put("total is =>") ; put(neightbors.size) ;
  --new_line ;
  RETURN neightbors;
END surrounding_positions;

FUNCTION useless(limit_depth : IN integer) RETURN boolean IS
BEGIN
  IF NOT (limit_depth > wg_limit_size) THEN
    RETURN false;
  ELSE
    RETURN true;
  END IF;
END useless;

FUNCTION usable(path   : path_type;
                depth  : board_range;
                player : IN player_type) RETURN boolean IS
BEGIN
  FOR member IN 2 .. depth - 1 LOOP
    FOR index IN 1 .. line_size LOOP
      IF path(member) = start(player, index) OR path(member) =
       ending(player, index) THEN
        RETURN true;
```

```
      END IF;
    END LOOP;
  END LOOP;
  FOR member IN 1 .. depth - 1 LOOP
    FOR next IN member + 2 .. depth LOOP
      IF is_connected(path(member), path(next)) THEN
        RETURN true;
      END IF;
    END LOOP;
  END LOOP;
  RETURN false;
END usable;

PROCEDURE create_paths(player   : IN player_type;
                       position : IN board_range;
                       depth    : IN board_range) IS
  successors : list;
BEGIN
  IF check_end_of_path(player, position) THEN
    IF NOT useless(depth - 1) AND NOT usable(path, depth - 1, player)
THEN
      wg_size(player) := wg_size(player) + 1;
      FOR index IN 1 .. depth - 1 LOOP
      --pos(player, path(index)).size := pos(player, path(index)).size
+ 1;
      --pos(player, path(index)).wg(pos(player, path(index)).size) :=
      --wg_size(player);
        wg(player, wg_size(player)).wg(index) := path(index);
      END LOOP;
      wg(player, wg_size(player)).size := depth - 1;
      IF max_wg(player) < (depth - 1) THEN
        max_wg(player) := depth - 1;
      END IF;
    END IF;
  ELSE
    IF NOT useless(depth - 1) AND NOT usable(path, depth - 1, player)
THEN
      successors := surrounding_positions(position);
      --put ("successors size is "); put(successors.size) ; new_line ;
      FOR index IN 1 .. successors.size LOOP
      --put("pos is ") ; put(position) ;
      --put(" next is =>"); put(successors.slots(index)) ;
        IF NOT occupied(successors.slots(index)) THEN
          occupied(successors.slots(index)) := true;
          path(depth) := successors.slots(index);
          create_paths(player, successors.slots(index), depth + 1);
          occupied(successors.slots(index)) := false;
          path(depth) := 0;
        END IF;
```

```
      END LOOP;
    END IF;
  END IF;
END create_paths;

PROCEDURE create_files IS
BEGIN
  put("separate(any_game)");
  new_line;
  put("function board_size  return natural is");
  new_line;
  put("begin");
  new_line;
  put("return  ");
  put(board_size, 4);
  new_line;
  put("end board_size ;");
  new_line;
  put("separate(wg_types)");
  new_line;
  put("function size_a return integer is ");
  new_line;
  put(" begin");
  new_line;
  put("return");
  put(wg_size(a));
  put(";");
  new_line;
  put("end size_a ;");
  new_line;
  put("separate(wg_types)");
  new_line;
  put("function size_b return integer is ");
  new_line;
  put(" begin");
  new_line;
  put("return");
  put(wg_size(b));
  put(";");
  new_line;
  put("end size_b ;");
  new_line;
  put("separate(any_game)");
  new_line;
  put("function initialize_win_groups_b return wg_table is");
  new_line;
  put("begin");
  new_line;
  put("return (");
```

```
new_line;
FOR index IN 1 .. wg_size(b) LOOP
  put("new wg_type'(");
  put(wg(b, index).size, 4);
  put(",(");
  FOR s IN 1 .. wg(b, index).size LOOP
    put(wg(b, index).wg(s), 3);
    IF s = wg(b, index).size THEN
      put(")");
    ELSE
      put(",");
    END IF;
  END LOOP;
  IF index = wg_size(b) THEN
    put("));");
  ELSE
    put("),");
    new_line;
  END IF;
END LOOP;
new_line;
put("end initialize_win_groups_b ;");
new_line;
put("separate(any_game)");
new_line;
put("function initialize_win_groups_a return wg_table is");
new_line;
put("begin");
new_line;
put("return (");
new_line;
FOR index IN 1 .. wg_size(a) LOOP
  put("new wg_type'(");
  put(wg(a, index).size, 4);
  put(",(");
  FOR s IN 1 .. wg(a, index).size LOOP
    put(wg(a, index).wg(s), 3);
    IF s = wg(a, index).size THEN
      put(")");
    ELSE
      put(",");
    END IF;
  END LOOP;
  IF index = wg_size(a) THEN
    put("));");
  ELSE
    put("),");
    new_line;
  END IF;
```

```
    END LOOP;
    new_line;
    put("end initialize_win_groups_a ;");
    new_line;
    new_line;
  END create_files;


BEGIN
-- Create start/end  up positions for the players
  FOR player IN player_type LOOP
    FOR pin IN max_range LOOP
      wg(player, pin) := null_wg;
    END LOOP;
    FOR pin IN 1 .. board_size LOOP
      pos(player, pin) := null_pos;
    END LOOP;
  END LOOP;
  start_b := -line_size + 1;
  end_a := line_size * (line_size - 1) + 1;
  IF board_size = 2 * (board_size / 2) THEN
    wg_limit_size := board_size / 2;
  ELSE
    wg_limit_size := board_size / 2 + 1;
  END IF;
  FOR index IN line_range LOOP
    start(a, index) := index;
    start_b := start_b + line_size;
    start(b, index) := start_b;
    ending(a, index) := end_a;
    ending(b, index) := index * line_size;
    end_a := end_a + 1;
  END LOOP;
  path := (board_range => 0);
  wg_size(a) := 0;
  wg_size(b) := 0;
  FOR player IN player_type LOOP
    FOR index IN line_range LOOP
      occupied(start(player, index)) := true;
      path(depth) := start(player, index);
      create_paths(player, start(player, index), depth + 1);
      occupied(start(player, index)) := false;
      path(depth) := 0;
    END LOOP;
  END LOOP;
  IF max_wg(a) > max_wg(b) THEN
    big := max_wg(a);
  ELSE
    big := max_wg(b);
```

```
  END IF;
  create_files;
END hex;
WITH integer_io;

USE  integer_io;

WITH text_io;
USE  text_io;


WITH wg_types;
USE  wg_types;

PROCEDURE gomoku IS
  TYPE    player_type IS (a, b);
  wg_limit_size              : integer;
  big_end                    : CONSTANT integer := 700;
  line_size                  : CONSTANT integer := 7;
  board_size                 : CONSTANT integer := line_size * line_size;
  SUBTYPE max_range   IS integer RANGE 1 .. big_end;
  wg_id                      : max_range;
  SUBTYPE board_range IS integer RANGE 1 .. line_size * line_size;
  wg_size                    : ARRAY(player_type) OF integer := (0, 0);
  max_wg                     : ARRAY(player_type) OF integer := (0, 0);
  big                        : integer;
  start_pos, next_pos_step : board_range;
  p_id                       : integer;
  TYPE pos_type IS RECORD
    size : integer := 0;
    wg   : wg_array(1 .. 350);
  END RECORD;

  TYPE wg_t IS RECORD
    size : integer;
    wg   : wg_array(1 .. 5);
  END RECORD;
  TYPE    p_acc       IS ACCESS pos_type;
  null_pos                   : pos_type         := (size => 0,
   wg => (1 .. 350 => 0));
  null_wg                    : wg_t             := (size => 5,
   wg => (1 .. 5 => 0));
  pos                        : ARRAY(player_type, board_range) OF p_acc :=
   (player_type => (board_range => NEW pos_type));
  wg                         : ARRAY(player_type, max_range) OF wg_t :=
   (player_type => (max_range => null_wg));
  SUBTYPE list_range  IS integer RANGE 0 .. 6;
  TYPE    slots_type  IS ARRAY(1 .. 6) OF board_range;
  TYPE list IS RECORD
```

```
  size  : list_range;
  slots : slots_type;
END RECORD;

FUNCTION linec(a : IN board_range) RETURN integer IS
BEGIN
  IF (a REM line_size) /= 0 THEN
    RETURN (a / line_size + 1);
  ELSE
    RETURN (a / line_size);
  END IF;
END linec;

PROCEDURE create_files IS
BEGIN
  put("separate(wg_types)");
  new_line;
  put("function size_a return integer is ");
  new_line;
  put(" begin");
  new_line;
  put("return");
  put(wg_size(a));
  put(";");
  new_line;
  put("end size_a ;");
  new_line;
  put("separate(wg_types)");
  new_line;
  put("function size_b return integer is ");
  new_line;
  put(" begin");
  new_line;
  put("return");
  put(wg_size(b));
  put(";");
  new_line;
  put("end size_b ;");
  new_line;
  put("separate(hex_game)");
  new_line;
  put("function initialize_win_groups_b return wg_table is");
  new_line;
  put("begin");
  new_line;
  put("return (");
  new_line;
  FOR index IN 1 .. wg_size(b) LOOP
    put("new wg_type'(");
```

```
  put(wg(b, index).size, 4);
  put(",(");
  FOR s IN 1 .. wg(b, index).size LOOP
    put(s, 4);
    put("=>");
    put(wg(b, index).wg(s));
    IF s = wg(b, index).size THEN
      put(")");
    ELSE
      put(",");
    END IF;
  END LOOP;
  IF index = wg_size(b) THEN
    put("));");
  ELSE
    put("),");
    new_line;
  END IF;
END LOOP;
new_line;
new_line;
put("separate(hex_game)");
new_line;
put("function initialize_win_groups_a return wg_table is");
new_line;
put("begin");
new_line;
put("return (");
new_line;
FOR index IN 1 .. wg_size(a) LOOP
  put("new wg_type'(");
  put(wg(a, index).size, 4);
  put(",(");
  FOR s IN 1 .. wg(a, index).size LOOP
    put(s, 4);
    put("=>");
    put(wg(a, index).wg(s));
    IF s = wg(a, index).size THEN
      put(")");
    ELSE
      put(",");
    END IF;
  END LOOP;
  IF index = wg_size(a) THEN
    put("));");
  ELSE
    put("),");
    new_line;
  END IF;
```

```
    END LOOP;
    new_line;
  END create_files;

BEGIN
  max_wg(a) := 5;
  max_wg(b) := 5;

  -- horizontal win-groups computation
  next_pos_step := 1;
  FOR line IN 1 .. line_size LOOP
    start_pos := (line - 1) * line_size + 1;
    FOR column IN 1 .. (line_size - 5 + 1) LOOP
      p_id := start_pos + column - 1;
      wg_size(a) := wg_size(a) + 1;
      wg_id := wg_size(a);
      FOR index IN 1 .. 5 LOOP
        wg(a, wg_id).wg(index) := p_id;
        pos(a, p_id).size := pos(a, p_id).size + 1;
        pos(a, p_id).wg(pos(a, p_id).size) := wg_id;
        p_id := p_id + next_pos_step;
      END LOOP;
    END LOOP;
  END LOOP;
                                                                    --   vertical
win-groups
                                                                  -- computation
  next_pos_step := line_size;
  FOR column IN 1 .. line_size LOOP
    start_pos := column;
    FOR line IN 1 .. (line_size - 5 + 1) LOOP
      p_id := (line - 1) * line_size + start_pos;
      wg_size(a) := wg_size(a) + 1;
      wg_id := wg_size(a);
      FOR index IN 1 .. 5 LOOP
        wg(a, wg_id).wg(index) := p_id;
        pos(a, p_id).size := pos(a, p_id).size + 1;
        pos(a, p_id).wg(pos(a, p_id).size) := wg_id;
        p_id := p_id + next_pos_step;
      END LOOP;
    END LOOP;
  END LOOP;
                                                                    --   diagonal
win-groups
                                                                  -- form left to
right
  next_pos_step := line_size + 1;
  FOR line IN 1 .. (line_size - 5 + 1) LOOP
    start_pos := (line - 1) * line_size + 1;
```

```
    FOR column IN 1 .. (line_size - 5 + 1) LOOP
      p_id := start_pos + column - 1;
      wg_size(a) := wg_size(a) + 1;
      wg_id := wg_size(a);
      FOR index IN 1 .. 5 LOOP
        wg(a, wg_id).wg(index) := p_id;
        pos(a, p_id).size := pos(a, p_id).size + 1;
        pos(a, p_id).wg(pos(a, p_id).size) := wg_id;
        p_id := p_id + next_pos_step;
      END LOOP;
    END LOOP;
  END LOOP;
  next_pos_step := line_size - 1;
  FOR line IN 1 .. (line_size - 5 + 1) LOOP
    start_pos := line * line_size;
    FOR column IN 1 .. (line_size - 5 + 1) LOOP
      p_id := start_pos - column + 1;
      wg_size(a) := wg_size(a) + 1;
      wg_id := wg_size(a);
      FOR index IN 1 .. 5 LOOP
        wg(a, wg_id).wg(index) := p_id;
        pos(a, p_id).size := pos(a, p_id).size + 1;
        pos(a, p_id).wg(pos(a, p_id).size) := wg_id;
        p_id := p_id + next_pos_step;
      END LOOP;
    END LOOP;
  END LOOP;
  wg_size(b) := wg_size(a);
  FOR index IN 1 .. wg_size(a) LOOP
    wg(b, index) := wg(a, index);
  END LOOP;
  FOR index IN 1 .. board_size LOOP
    pos(b, index) := pos(a, index);
  END LOOP;
  IF max_wg(a) > max_wg(b) THEN
    big := max_wg(a);
  ELSE
    big := max_wg(b);
  END IF;
  create_files;
END gomoku;
```

APPENDIX B

```
package board is

type wg_array is array(integer range <>) of integer ;
type wg_type(size : integer := 0) is record
     wg : wg_array(1.. size) ;
     end record ;

type wg_access is access wg_type ;
type wg_table is array(integer range <>) of wg_access ;

 number_of_positions : constant natural := 49 ;
 type values is (player_a, player_b, empty, not_available ) ;
 subtype board_range is integer range 1..number_of_positions ;

 wga : constant wg_table(1..60) := (
new wg_type'(5,(  1,  2, 3,  4,  5)),
new wg_type'(5,(  2,  3, 4,  5,  6)),
new wg_type'(5,(  3,  4, 5,  6,  7)),
new wg_type'(5,(  8,  9, 10, 11, 12)),
new wg_type'(5,(  9, 10, 11, 12, 13)),
new wg_type'(5,( 10, 11, 12, 13, 14)),
new wg_type'(5,( 15, 16, 17, 18, 19)),
new wg_type'(5,( 16, 17, 18, 19, 20)),
new wg_type'(5,( 17, 18, 19, 20, 21)),
new wg_type'(5,( 22, 23, 24, 25, 26)),
new wg_type'(5,( 23, 24, 25, 26, 27)),
new wg_type'(5,( 24, 25, 26, 27, 28)),
new wg_type'(5,( 29, 30, 31, 32, 33)),
new wg_type'(5,( 30, 31, 32, 33, 34)),
new wg_type'(5,( 31, 32, 33, 34, 35)),
new wg_type'(5,( 36, 37, 38, 39, 40)),
new wg_type'(5,( 37, 38, 39, 40, 41)),
new wg_type'(5,( 38, 39, 40, 41, 42)),
new wg_type'(5,( 43, 44, 45, 46, 47)),
new wg_type'(5,( 44, 45, 46, 47, 48)),
new wg_type'(5,( 45, 46, 47, 48, 49)),
new wg_type'(5,(  1,  8, 15, 22, 29)),
new wg_type'(5,(  8, 15, 22, 29, 36)),
new wg_type'(5,( 15, 22, 29, 36, 43)),
new wg_type'(5,(  2,  9, 16, 23, 30)),
new wg_type'(5,(  9, 16, 23, 30, 37)),
new wg_type'(5,( 16, 23, 30, 37, 44)),
new wg_type'(5,(  3, 10, 17, 24, 31)),
new wg_type'(5,( 10, 17, 24, 31, 38)),
new wg_type'(5,( 17, 24, 31, 38, 45)),
new wg_type'(5,(  4, 11, 18, 25, 32)),
new wg_type'(5,( 11, 18, 25, 32, 39)),
new wg_type'(5,( 18, 25, 32, 39, 46)),
new wg_type'(5,(  5, 12, 19, 26, 33)),
```

```
new wg_type'(5,( 12, 19, 26, 33, 40)),
new wg_type'(5,( 19, 26, 33, 40, 47)),
new wg_type'(5,(  6, 13, 20, 27, 34)),
new wg_type'(5,( 13, 20, 27, 34, 41)),
new wg_type'(5,( 20, 27, 34, 41, 48)),
new wg_type'(5,(  7, 14, 21, 28, 35)),
new wg_type'(5,( 14, 21, 28, 35, 42)),
new wg_type'(5,( 21, 28, 35, 42, 49)),
new wg_type'(5,(  1,  9, 17, 25, 33)),
new wg_type'(5,(  2, 10, 18, 26, 34)),
new wg_type'(5,(  3, 11, 19, 27, 35)),
new wg_type'(5,(  8, 16, 24, 32, 40)),
new wg_type'(5,(  9, 17, 25, 33, 41)),
new wg_type'(5,( 10, 18, 26, 34, 42)),
new wg_type'(5,( 15, 23, 31, 39, 47)),
new wg_type'(5,( 16, 24, 32, 40, 48)),
new wg_type'(5,( 17, 25, 33, 41, 49)),
new wg_type'(5,(  7, 13, 19, 25, 31)),
new wg_type'(5,(  6, 12, 18, 24, 30)),
new wg_type'(5,(  5, 11, 17, 23, 29)),
new wg_type'(5,( 14, 20, 26, 32, 38)),
new wg_type'(5,( 13, 19, 25, 31, 37)),
new wg_type'(5,( 12, 18, 24, 30, 36)),
new wg_type'(5,( 21, 27, 33, 39, 45)),
new wg_type'(5,( 20, 26, 32, 38, 44)),
new wg_type'(5,( 19, 25, 31, 37, 43)));

wgb : constant wg_table(1..60) := (
new wg_type'(5,(  1,  2,  3,  4,  5)),
new wg_type'(5,(  2,  3,  4,  5,  6)),
new wg_type'(5,(  3,  4,  5,  6,  7)),
new wg_type'(5,(  8,  9, 10, 11, 12)),
new wg_type'(5,(  9, 10, 11, 12, 13)),
new wg_type'(5,( 10, 11, 12, 13, 14)),
new wg_type'(5,( 15, 16, 17, 18, 19)),
new wg_type'(5,( 16, 17, 18, 19, 20)),
new wg_type'(5,( 17, 18, 19, 20, 21)),
new wg_type'(5,( 22, 23, 24, 25, 26)),
new wg_type'(5,( 23, 24, 25, 26, 27)),
new wg_type'(5,( 24, 25, 26, 27, 28)),
new wg_type'(5,( 29, 30, 31, 32, 33)),
new wg_type'(5,( 30, 31, 32, 33, 34)),
new wg_type'(5,( 31, 32, 33, 34, 35)),
new wg_type'(5,( 36, 37, 38, 39, 40)),
new wg_type'(5,( 37, 38, 39, 40, 41)),
new wg_type'(5,( 38, 39, 40, 41, 42)),
new wg_type'(5,( 43, 44, 45, 46, 47)),
new wg_type'(5,( 44, 45, 46, 47, 48)),
new wg_type'(5,( 45, 46, 47, 48, 49)),
```

```
new wg_type'(5,(  1,  8, 15, 22, 29)),
new wg_type'(5,(  8, 15, 22, 29, 36)),
new wg_type'(5,( 15, 22, 29, 36, 43)),
new wg_type'(5,(  2,  9, 16, 23, 30)),
new wg_type'(5,(  9, 16, 23, 30, 37)),
new wg_type'(5,( 16, 23, 30, 37, 44)),
new wg_type'(5,(  3, 10, 17, 24, 31)),
new wg_type'(5,( 10, 17, 24, 31, 38)),
new wg_type'(5,( 17, 24, 31, 38, 45)),
new wg_type'(5,(  4, 11, 18, 25, 32)),
new wg_type'(5,( 11, 18, 25, 32, 39)),
new wg_type'(5,( 18, 25, 32, 39, 46)),
new wg_type'(5,(  5, 12, 19, 26, 33)),
new wg_type'(5,( 12, 19, 26, 33, 40)),
new wg_type'(5,( 19, 26, 33, 40, 47)),
new wg_type'(5,(  6, 13, 20, 27, 34)),
new wg_type'(5,( 13, 20, 27, 34, 41)),
new wg_type'(5,( 20, 27, 34, 41, 48)),
new wg_type'(5,(  7, 14, 21, 28, 35)),
new wg_type'(5,( 14, 21, 28, 35, 42)),
new wg_type'(5,( 21, 28, 35, 42, 49)),
new wg_type'(5,(  1,  9, 17, 25, 33)),
new wg_type'(5,(  2, 10, 18, 26, 34)),
new wg_type'(5,(  3, 11, 19, 27, 35)),
new wg_type'(5,(  8, 16, 24, 32, 40)),
new wg_type'(5,(  9, 17, 25, 33, 41)),
new wg_type'(5,( 10, 18, 26, 34, 42)),
new wg_type'(5,( 15, 23, 31, 39, 47)),
new wg_type'(5,( 16, 24, 32, 40, 48)),
new wg_type'(5,( 17, 25, 33, 41, 49)),
new wg_type'(5,(  7, 13, 19, 25, 31)),
new wg_type'(5,(  6, 12, 18, 24, 30)),
new wg_type'(5,(  5, 11, 17, 23, 29)),
new wg_type'(5,( 14, 20, 26, 32, 38)),
new wg_type'(5,( 13, 19, 25, 31, 37)),
new wg_type'(5,( 12, 18, 24, 30, 36)),
new wg_type'(5,( 21, 27, 33, 39, 45)),
new wg_type'(5,( 20, 26, 32, 38, 44)),
new wg_type'(5,( 19, 25, 31, 37, 43)));
end board ;



--- Package Board for game Hex/4

package board is

type wg_array is array(integer range <>) of integer ;
type wg_type(size : integer := 0) is record
```

```
wg : wg_array(1.. size) ;
end record ;

type wg_access is access wg_type ;
type wg_table is array(integer range <>) of wg_access ;

 number_of_positions : constant natural := 16 ;
 type values is (player_a, player_b, empty, not_available ) ;
 subtype board_range is integer range 1..number_of_positions ;

 wga : constant wg_table(1..54) :=
(new wg_type'(7,(  1,  5,  6,  7,  8, 12, 15)),
new wg_type'(7,(  1,  5,  6,  7,  8, 12, 16)),
new wg_type'(7,(  1,  5,  6,  7, 11, 12, 16)),
new wg_type'(6,(  1,  5,  6,  7, 11, 14)),
new wg_type'(6,(  1,  5,  6,  7, 11, 15)),
new wg_type'(7,(  1,  5,  6, 10, 11, 12, 16)),
new wg_type'(6,(  1,  5,  6, 10, 11, 15)),
new wg_type'(5,(  1,  5,  6, 10, 13)),
new wg_type'(5,(  1,  5,  6, 10, 14)),
new wg_type'(7,(  1,  5,  9, 10, 11, 12, 16)),
new wg_type'(6,(  1,  5,  9, 10, 11, 15)),
new wg_type'(8,(  1,  5,  9, 10,  7,  8, 12, 15)),
new wg_type'(8,(  1,  5,  9, 10,  7,  8, 12, 16)),
new wg_type'(5,(  1,  5,  9, 10, 14)),
new wg_type'(4,(  1,  5,  9, 13)),
new wg_type'(7,(  2,  5,  9, 10, 11, 12, 16)),
new wg_type'(6,(  2,  5,  9, 10, 11, 15)),
new wg_type'(8,(  2,  5,  9, 10,  7,  8, 12, 15)),
new wg_type'(8,(  2,  5,  9, 10,  7,  8, 12, 16)),
new wg_type'(5,(  2,  5,  9, 10, 14)),
new wg_type'(4,(  2,  5,  9, 13)),
new wg_type'(6,(  2,  6,  7,  8, 12, 15)),
new wg_type'(6,(  2,  6,  7,  8, 12, 16)),
new wg_type'(6,(  2,  6,  7, 11, 12, 16)),
new wg_type'(5,(  2,  6,  7, 11, 14)),
new wg_type'(5,(  2,  6,  7, 11, 15)),
new wg_type'(4,(  2,  6,  9, 13)),
new wg_type'(6,(  2,  6, 10, 11, 12, 16)),
new wg_type'(5,(  2,  6, 10, 11, 15)),
new wg_type'(4,(  2,  6, 10, 13)),
new wg_type'(4,(  2,  6, 10, 14)),
new wg_type'(4,(  3,  6,  9, 13)),
new wg_type'(6,(  3,  6, 10, 11, 12, 16)),
new wg_type'(5,(  3,  6, 10, 11, 15)),
new wg_type'(4,(  3,  6, 10, 13)),
new wg_type'(4,(  3,  6, 10, 14)),
new wg_type'(5,(  3,  7,  8, 12, 15)),
new wg_type'(5,(  3,  7,  8, 12, 16)),
```

```
new wg_type'(4,(  3,  7, 10, 13)),
new wg_type'(4,(  3,  7, 10, 14)),
new wg_type'(5,(  3,  7, 11, 12, 16)),
new wg_type'(4,(  3,  7, 11, 14)),
new wg_type'(4,(  3,  7, 11, 15)),
new wg_type'(5,(  4,  7,  6,  9, 13)),
new wg_type'(4,(  4,  7, 10, 13)),
new wg_type'(4,(  4,  7, 10, 14)),
new wg_type'(5,(  4,  7, 11, 12, 16)),
new wg_type'(4,(  4,  7, 11, 14)),
new wg_type'(4,(  4,  7, 11, 15)),
new wg_type'(5,(  4,  8, 11, 10, 13)),
new wg_type'(4,(  4,  8, 11, 14)),
new wg_type'(4,(  4,  8, 11, 15)),
new wg_type'(4,(  4,  8, 12, 15)),
new wg_type'(4,(  4,  8, 12, 16)));
 wgb : wg_table(1..54) :=
(
new wg_type'(4,(  1,  2,  3,  4)),
new wg_type'(5,(  1,  2,  3,  7,  8)),
new wg_type'(8,(  1,  2,  3,  7, 10, 14, 15, 16)),
new wg_type'(8,(  1,  2,  3,  7, 10, 14, 15, 12)),
new wg_type'(6,(  1,  2,  3,  7, 11, 12)),
new wg_type'(7,(  1,  2,  3,  7, 11, 15, 16)),
new wg_type'(5,(  1,  2,  6,  7,  8)),
new wg_type'(5,(  1,  2,  6,  7,  4)),
new wg_type'(6,(  1,  2,  6,  7, 11, 12)),
new wg_type'(7,(  1,  2,  6,  7, 11, 15, 16)),
new wg_type'(6,(  1,  2,  6, 10, 11, 12)),
new wg_type'(6,(  1,  2,  6, 10, 11,  8)),
new wg_type'(7,(  1,  2,  6, 10, 11, 15, 16)),
new wg_type'(7,(  1,  2,  6, 10, 14, 15, 16)),
new wg_type'(7,(  1,  2,  6, 10, 14, 15, 12)),
new wg_type'(4,(  5,  6,  7,  8)),
new wg_type'(4,(  5,  6,  7,  4)),
new wg_type'(5,(  5,  6,  7, 11, 12)),
new wg_type'(6,(  5,  6,  7, 11, 15, 16)),
new wg_type'(4,(  5,  6,  3,  4)),
new wg_type'(5,(  5,  6, 10, 11, 12)),
new wg_type'(5,(  5,  6, 10, 11,  8)),
new wg_type'(6,(  5,  6, 10, 11, 15, 16)),
new wg_type'(6,(  5,  6, 10, 14, 15, 16)),
new wg_type'(6,(  5,  6, 10, 14, 15, 12)),
new wg_type'(4,(  5,  2,  3,  4)),
new wg_type'(5,(  5,  2,  3,  7,  8)),
new wg_type'(8,(  5,  2,  3,  7, 10, 14, 15, 16)),
new wg_type'(8,(  5,  2,  3,  7, 10, 14, 15, 12)),
new wg_type'(6,(  5,  2,  3,  7, 11, 12)),
new wg_type'(7,(  5,  2,  3,  7, 11, 15, 16)),
```

```
new wg_type'(4,(  9, 10, 11, 12)),
new wg_type'(4,(  9, 10, 11,  8)),
new wg_type'(5,(  9, 10, 11, 15, 16)),
new wg_type'(4,(  9, 10,  7,  8)),
new wg_type'(4,(  9, 10,  7,  4)),
new wg_type'(5,(  9, 10, 14, 15, 16)),
new wg_type'(5,(  9, 10, 14, 15, 12)),
new wg_type'(4,(  9,  6,  7,  8)),
new wg_type'(4,(  9,  6,  7,  4)),
new wg_type'(5,(  9,  6,  7, 11, 12)),
new wg_type'(6,(  9,  6,  7, 11, 15, 16)),
new wg_type'(4,(  9,  6,  3,  4)),
new wg_type'(4,( 13, 14, 15, 16)),
new wg_type'(4,( 13, 14, 15, 12)),
new wg_type'(4,( 13, 14, 11, 12)),
new wg_type'(4,( 13, 14, 11,  8)),
new wg_type'(5,( 13, 14, 11,  7,  4)),
new wg_type'(4,( 13, 10, 11, 12)),
new wg_type'(4,( 13, 10, 11,  8)),
new wg_type'(5,( 13, 10, 11, 15, 16)),
new wg_type'(4,( 13, 10,  7,  8)),
new wg_type'(4,( 13, 10,  7,  4)),
new wg_type'(5,( 13, 10,  6,  3,  4)));
end board ;
```

```
--- Package Board for game Hex/5

package board is

type wg_array is array(integer range <>) of integer ;
type wg_type(size : integer := 0) is record
```

```
wg : wg_array(1.. size) ;
end record ;

type wg_access is access wg_type ;
type wg_table is array(integer range <>) of wg_access ;

 number_of_positions : constant natural := 25 ;
 type values is (player_a, player_b, empty, not_available ) ;
 subtype board_range is integer range 1..number_of_positions ;

 wga : constant wg_table(1..365) :=
(
new wg_type'(5,(  1,  2,  3,  4,  5)),
new wg_type'(6,(  1,  2,  3,  4,  9, 10)),
new wg_type'(  11,(  1,  2,  3,  4,  9, 13, 17, 22, 23, 24, 25)),
new wg_type'(  11,(  1,  2,  3,  4,  9, 13, 17, 22, 23, 24, 20)),
new wg_type'(  11,(  1,  2,  3,  4,  9, 13, 17, 22, 23, 19, 20)),
new wg_type'(  11,(  1,  2,  3,  4,  9, 13, 17, 22, 23, 19, 15)),
new wg_type'(9,(  1,  2,  3,  4,  9, 13, 18, 19, 20)),
new wg_type'(9,(  1,  2,  3,  4,  9, 13, 18, 19, 15)),
new wg_type'(  10,(  1,  2,  3,  4,  9, 13, 18, 19, 24, 25)),
new wg_type'(  10,(  1,  2,  3,  4,  9, 13, 18, 23, 24, 25)),
new wg_type'(  10,(  1,  2,  3,  4,  9, 13, 18, 23, 24, 20)),
new wg_type'(7,(  1,  2,  3,  4,  9, 14, 15)),
new wg_type'(  10,(  1,  2,  3,  4,  9, 14, 18, 23, 24, 25)),
new wg_type'(  10,(  1,  2,  3,  4,  9, 14, 18, 23, 24, 20)),
new wg_type'(8,(  1,  2,  3,  4,  9, 14, 19, 20)),
new wg_type'(9,(  1,  2,  3,  4,  9, 14, 19, 24, 25)),
new wg_type'(6,(  1,  2,  3,  8,  9, 10)),
new wg_type'(6,(  1,  2,  3,  8,  9,  5)),
new wg_type'(7,(  1,  2,  3,  8,  9, 14, 15)),
new wg_type'(  10,(  1,  2,  3,  8,  9, 14, 18, 23, 24, 25)),
new wg_type'(  10,(  1,  2,  3,  8,  9, 14, 18, 23, 24, 20)),
new wg_type'(8,(  1,  2,  3,  8,  9, 14, 19, 20)),
new wg_type'(9,(  1,  2,  3,  8,  9, 14, 19, 24, 25)),
new wg_type'(9,(  1,  2,  3,  8, 12, 17, 18, 19, 20)),
new wg_type'(9,(  1,  2,  3,  8, 12, 17, 18, 19, 15)),
new wg_type'(  10,(  1,  2,  3,  8, 12, 17, 18, 19, 24, 25)),
new wg_type'(9,(  1,  2,  3,  8, 12, 17, 18, 14, 15)),
new wg_type'(9,(  1,  2,  3,  8, 12, 17, 18, 14, 10)),
new wg_type'(  10,(  1,  2,  3,  8, 12, 17, 18, 23, 24, 25)),
new wg_type'(  10,(  1,  2,  3,  8, 12, 17, 18, 23, 24, 20)),
new wg_type'(  10,(  1,  2,  3,  8, 12, 17, 22, 23, 24, 25)),
new wg_type'(  10,(  1,  2,  3,  8, 12, 17, 22, 23, 24, 20)),
new wg_type'(  10,(  1,  2,  3,  8, 12, 17, 22, 23, 19, 20)),
new wg_type'(  10,(  1,  2,  3,  8, 12, 17, 22, 23, 19, 15)),
new wg_type'(  11,(  1,  2,  3,  8, 12, 17, 22, 23, 19, 14, 10)),
new wg_type'(7,(  1,  2,  3,  8, 13, 14, 15)),
new wg_type'(7,(  1,  2,  3,  8, 13, 14, 10)),
```

```
new wg_type'(8,(  1,  2,  3,  8, 13, 14, 19, 20)),
new wg_type'(9,(  1,  2,  3,  8, 13, 14, 19, 24, 25)),
new wg_type'(  10,(  1,  2,  3,  8, 13, 17, 22, 23, 24, 25)),
new wg_type'(  10,(  1,  2,  3,  8, 13, 17, 22, 23, 24, 20)),
new wg_type'(  10,(  1,  2,  3,  8, 13, 17, 22, 23, 19, 20)),
new wg_type'(  10,(  1,  2,  3,  8, 13, 17, 22, 23, 19, 15)),
new wg_type'(8,(  1,  2,  3,  8, 13, 18, 19, 20)),
new wg_type'(8,(  1,  2,  3,  8, 13, 18, 19, 15)),
new wg_type'(9,(  1,  2,  3,  8, 13, 18, 19, 24, 25)),
new wg_type'(9,(  1,  2,  3,  8, 13, 18, 23, 24, 25)),
new wg_type'(9,(  1,  2,  3,  8, 13, 18, 23, 24, 20)),
new wg_type'(6,(  1,  2,  7,  8,  9, 10)),
new wg_type'(6,(  1,  2,  7,  8,  9,  5)),
new wg_type'(7,(  1,  2,  7,  8,  9, 14, 15)),
new wg_type'(  10,(  1,  2,  7,  8,  9, 14, 18, 23, 24, 25)),
new wg_type'(  10,(  1,  2,  7,  8,  9, 14, 18, 23, 24, 20)),
new wg_type'(8,(  1,  2,  7,  8,  9, 14, 19, 20)),
new wg_type'(9,(  1,  2,  7,  8,  9, 14, 19, 24, 25)),
new wg_type'(6,(  1,  2,  7,  8,  4,  5)),
new wg_type'(7,(  1,  2,  7,  8, 13, 14, 15)),
new wg_type'(7,(  1,  2,  7,  8, 13, 14, 10)),
new wg_type'(8,(  1,  2,  7,  8, 13, 14, 19, 20)),
new wg_type'(9,(  1,  2,  7,  8, 13, 14, 19, 24, 25)),
new wg_type'(  10,(  1,  2,  7,  8, 13, 17, 22, 23, 24, 25)),
new wg_type'(  10,(  1,  2,  7,  8, 13, 17, 22, 23, 24, 20)),
new wg_type'(  10,(  1,  2,  7,  8, 13, 17, 22, 23, 19, 20)),
new wg_type'(  10,(  1,  2,  7,  8, 13, 17, 22, 23, 19, 15)),
new wg_type'(8,(  1,  2,  7,  8, 13, 18, 19, 20)),
new wg_type'(8,(  1,  2,  7,  8, 13, 18, 19, 15)),
new wg_type'(9,(  1,  2,  7,  8, 13, 18, 19, 24, 25)),
new wg_type'(9,(  1,  2,  7,  8, 13, 18, 23, 24, 25)),
new wg_type'(9,(  1,  2,  7,  8, 13, 18, 23, 24, 20)),
new wg_type'(7,(  1,  2,  7, 12, 13, 14, 15)),
new wg_type'(7,(  1,  2,  7, 12, 13, 14, 10)),
new wg_type'(8,(  1,  2,  7, 12, 13, 14, 19, 20)),
new wg_type'(9,(  1,  2,  7, 12, 13, 14, 19, 24, 25)),
new wg_type'(7,(  1,  2,  7, 12, 13,  9, 10)),
new wg_type'(7,(  1,  2,  7, 12, 13,  9,  5)),
new wg_type'(8,(  1,  2,  7, 12, 13, 18, 19, 20)),
new wg_type'(8,(  1,  2,  7, 12, 13, 18, 19, 15)),
new wg_type'(9,(  1,  2,  7, 12, 13, 18, 19, 24, 25)),
new wg_type'(9,(  1,  2,  7, 12, 13, 18, 23, 24, 25)),
new wg_type'(9,(  1,  2,  7, 12, 13, 18, 23, 24, 20)),
new wg_type'(8,(  1,  2,  7, 12, 17, 18, 19, 20)),
new wg_type'(8,(  1,  2,  7, 12, 17, 18, 19, 15)),
new wg_type'(9,(  1,  2,  7, 12, 17, 18, 19, 24, 25)),
new wg_type'(8,(  1,  2,  7, 12, 17, 18, 14, 15)),
new wg_type'(8,(  1,  2,  7, 12, 17, 18, 14, 10)),
new wg_type'(9,(  1,  2,  7, 12, 17, 18, 14,  9,  5)),
```

```
new wg_type'(9,(  1,  2,  7, 12, 17, 18, 23, 24, 25)),
new wg_type'(9,(  1,  2,  7, 12, 17, 18, 23, 24, 20)),
new wg_type'(9,(  1,  2,  7, 12, 17, 22, 23, 24, 25)),
new wg_type'(9,(  1,  2,  7, 12, 17, 22, 23, 24, 20)),
new wg_type'(9,(  1,  2,  7, 12, 17, 22, 23, 19, 20)),
new wg_type'(9,(  1,  2,  7, 12, 17, 22, 23, 19, 15)),
new wg_type'(  10,(  1,  2,  7, 12, 17, 22, 23, 19, 14, 10)),
new wg_type'(  11,(  1,  2,  7, 12, 17, 22, 23, 19, 14,  9,  5)),
new wg_type'(5,(  6,  7,  8,  9, 10)),
new wg_type'(5,(  6,  7,  8,  9,  5)),
new wg_type'(6,(  6,  7,  8,  9, 14, 15)),
new wg_type'(9,(  6,  7,  8,  9, 14, 18, 23, 24, 25)),
new wg_type'(9,(  6,  7,  8,  9, 14, 18, 23, 24, 20)),
new wg_type'(7,(  6,  7,  8,  9, 14, 19, 20)),
new wg_type'(8,(  6,  7,  8,  9, 14, 19, 24, 25)),
new wg_type'(5,(  6,  7,  8,  4,  5)),
new wg_type'(6,(  6,  7,  8, 13, 14, 15)),
new wg_type'(6,(  6,  7,  8, 13, 14, 10)),
new wg_type'(7,(  6,  7,  8, 13, 14, 19, 20)),
new wg_type'(8,(  6,  7,  8, 13, 14, 19, 24, 25)),
new wg_type'(9,(  6,  7,  8, 13, 17, 22, 23, 24, 25)),
new wg_type'(9,(  6,  7,  8, 13, 17, 22, 23, 24, 20)),
new wg_type'(9,(  6,  7,  8, 13, 17, 22, 23, 19, 20)),
new wg_type'(9,(  6,  7,  8, 13, 17, 22, 23, 19, 15)),
new wg_type'(7,(  6,  7,  8, 13, 18, 19, 20)),
new wg_type'(7,(  6,  7,  8, 13, 18, 19, 15)),
new wg_type'(8,(  6,  7,  8, 13, 18, 19, 24, 25)),
new wg_type'(8,(  6,  7,  8, 13, 18, 23, 24, 25)),
new wg_type'(8,(  6,  7,  8, 13, 18, 23, 24, 20)),
new wg_type'(5,(  6,  7,  3,  4,  5)),
new wg_type'(6,(  6,  7,  3,  4,  9, 10)),
new wg_type'(  11,(  6,  7,  3,  4,  9, 13, 17, 22, 23, 24, 25)),
new wg_type'(  11,(  6,  7,  3,  4,  9, 13, 17, 22, 23, 24, 20)),
new wg_type'(  11,(  6,  7,  3,  4,  9, 13, 17, 22, 23, 19, 20)),
new wg_type'(  11,(  6,  7,  3,  4,  9, 13, 17, 22, 23, 19, 15)),
new wg_type'(9,(  6,  7,  3,  4,  9, 13, 18, 19, 20)),
new wg_type'(9,(  6,  7,  3,  4,  9, 13, 18, 19, 15)),
new wg_type'(  10,(  6,  7,  3,  4,  9, 13, 18, 19, 24, 25)),
new wg_type'(  10,(  6,  7,  3,  4,  9, 13, 18, 23, 24, 25)),
new wg_type'(  10,(  6,  7,  3,  4,  9, 13, 18, 23, 24, 20)),
new wg_type'(7,(  6,  7,  3,  4,  9, 14, 15)),
new wg_type'(  10,(  6,  7,  3,  4,  9, 14, 18, 23, 24, 25)),
new wg_type'(  10,(  6,  7,  3,  4,  9, 14, 18, 23, 24, 20)),
new wg_type'(8,(  6,  7,  3,  4,  9, 14, 19, 20)),
new wg_type'(9,(  6,  7,  3,  4,  9, 14, 19, 24, 25)),
new wg_type'(6,(  6,  7, 12, 13, 14, 15)),
new wg_type'(6,(  6,  7, 12, 13, 14, 10)),
new wg_type'(7,(  6,  7, 12, 13, 14, 19, 20)),
new wg_type'(8,(  6,  7, 12, 13, 14, 19, 24, 25)),
```

```
new wg_type'(6,(  6,  7, 12, 13,  9, 10)),
new wg_type'(6,(  6,  7, 12, 13,  9,  5)),
new wg_type'(7,(  6,  7, 12, 13, 18, 19, 20)),
new wg_type'(7,(  6,  7, 12, 13, 18, 19, 15)),
new wg_type'(8,(  6,  7, 12, 13, 18, 19, 24, 25)),
new wg_type'(8,(  6,  7, 12, 13, 18, 23, 24, 25)),
new wg_type'(8,(  6,  7, 12, 13, 18, 23, 24, 20)),
new wg_type'(7,(  6,  7, 12, 17, 18, 19, 20)),
new wg_type'(7,(  6,  7, 12, 17, 18, 19, 15)),
new wg_type'(8,(  6,  7, 12, 17, 18, 19, 24, 25)),
new wg_type'(7,(  6,  7, 12, 17, 18, 14, 15)),
new wg_type'(7,(  6,  7, 12, 17, 18, 14, 10)),
new wg_type'(8,(  6,  7, 12, 17, 18, 14,  9,  5)),
new wg_type'(8,(  6,  7, 12, 17, 18, 23, 24, 25)),
new wg_type'(8,(  6,  7, 12, 17, 18, 23, 24, 20)),
new wg_type'(8,(  6,  7, 12, 17, 22, 23, 24, 25)),
new wg_type'(8,(  6,  7, 12, 17, 22, 23, 24, 20)),
new wg_type'(8,(  6,  7, 12, 17, 22, 23, 19, 20)),
new wg_type'(8,(  6,  7, 12, 17, 22, 23, 19, 15)),
new wg_type'(9,(  6,  7, 12, 17, 22, 23, 19, 14, 10)),
new wg_type'(  10,(  6,  7, 12, 17, 22, 23, 19, 14,  9,  5)),
new wg_type'(5,(  6,  2,  3,  4,  5)),
new wg_type'(6,(  6,  2,  3,  4,  9, 10)),
new wg_type'(  11,(  6,  2,  3,  4,  9, 13, 17, 22, 23, 24, 25)),
new wg_type'(  11,(  6,  2,  3,  4,  9, 13, 17, 22, 23, 24, 20)),
new wg_type'(  11,(  6,  2,  3,  4,  9, 13, 17, 22, 23, 19, 20)),
new wg_type'(  11,(  6,  2,  3,  4,  9, 13, 17, 22, 23, 19, 15)),
new wg_type'(9,(  6,  2,  3,  4,  9, 13, 18, 19, 20)),
new wg_type'(9,(  6,  2,  3,  4,  9, 13, 18, 19, 15)),
new wg_type'(  10,(  6,  2,  3,  4,  9, 13, 18, 19, 24, 25)),
new wg_type'(  10,(  6,  2,  3,  4,  9, 13, 18, 23, 24, 25)),
new wg_type'(  10,(  6,  2,  3,  4,  9, 13, 18, 23, 24, 20)),
new wg_type'(7,(  6,  2,  3,  4,  9, 14, 15)),
new wg_type'(  10,(  6,  2,  3,  4,  9, 14, 18, 23, 24, 25)),
new wg_type'(  10,(  6,  2,  3,  4,  9, 14, 18, 23, 24, 20)),
new wg_type'(8,(  6,  2,  3,  4,  9, 14, 19, 20)),
new wg_type'(9,(  6,  2,  3,  4,  9, 14, 19, 24, 25)),
new wg_type'(6,(  6,  2,  3,  8,  9, 10)),
new wg_type'(6,(  6,  2,  3,  8,  9,  5)),
new wg_type'(7,(  6,  2,  3,  8,  9, 14, 15)),
new wg_type'(  10,(  6,  2,  3,  8,  9, 14, 18, 23, 24, 25)),
new wg_type'(  10,(  6,  2,  3,  8,  9, 14, 18, 23, 24, 20)),
new wg_type'(8,(  6,  2,  3,  8,  9, 14, 19, 20)),
new wg_type'(9,(  6,  2,  3,  8,  9, 14, 19, 24, 25)),
new wg_type'(9,(  6,  2,  3,  8, 12, 17, 18, 19, 20)),
new wg_type'(9,(  6,  2,  3,  8, 12, 17, 18, 19, 15)),
new wg_type'(  10,(  6,  2,  3,  8, 12, 17, 18, 19, 24, 25)),
new wg_type'(9,(  6,  2,  3,  8, 12, 17, 18, 14, 15)),
new wg_type'(9,(  6,  2,  3,  8, 12, 17, 18, 14, 10)),
```

```
new wg_type'(  10,(  6,  2,  3,  8, 12, 17, 18, 23, 24, 25)),
new wg_type'(  10,(  6,  2,  3,  8, 12, 17, 18, 23, 24, 20)),
new wg_type'(  10,(  6,  2,  3,  8, 12, 17, 22, 23, 24, 25)),
new wg_type'(  10,(  6,  2,  3,  8, 12, 17, 22, 23, 24, 20)),
new wg_type'(  10,(  6,  2,  3,  8, 12, 17, 22, 23, 19, 20)),
new wg_type'(  10,(  6,  2,  3,  8, 12, 17, 22, 23, 19, 15)),
new wg_type'(  11,(  6,  2,  3,  8, 12, 17, 22, 23, 19, 14, 10)),
new wg_type'(7,(  6,  2,  3,  8, 13, 14, 15)),
new wg_type'(7,(  6,  2,  3,  8, 13, 14, 10)),
new wg_type'(8,(  6,  2,  3,  8, 13, 14, 19, 20)),
new wg_type'(9,(  6,  2,  3,  8, 13, 14, 19, 24, 25)),
new wg_type'(  10,(  6,  2,  3,  8, 13, 17, 22, 23, 24, 25)),
new wg_type'(  10,(  6,  2,  3,  8, 13, 17, 22, 23, 24, 20)),
new wg_type'(  10,(  6,  2,  3,  8, 13, 17, 22, 23, 19, 20)),
new wg_type'(  10,(  6,  2,  3,  8, 13, 17, 22, 23, 19, 15)),
new wg_type'(8,(  6,  2,  3,  8, 13, 18, 19, 20)),
new wg_type'(8,(  6,  2,  3,  8, 13, 18, 19, 15)),
new wg_type'(9,(  6,  2,  3,  8, 13, 18, 19, 24, 25)),
new wg_type'(9,(  6,  2,  3,  8, 13, 18, 23, 24, 25)),
new wg_type'(9,(  6,  2,  3,  8, 13, 18, 23, 24, 20)),
new wg_type'(5,( 11, 12, 13, 14, 15)),
new wg_type'(5,( 11, 12, 13, 14, 10)),
new wg_type'(6,( 11, 12, 13, 14, 19, 20)),
new wg_type'(7,( 11, 12, 13, 14, 19, 24, 25)),
new wg_type'(5,( 11, 12, 13,  9, 10)),
new wg_type'(5,( 11, 12, 13,  9,  5)),
new wg_type'(6,( 11, 12, 13, 18, 19, 20)),
new wg_type'(6,( 11, 12, 13, 18, 19, 15)),
new wg_type'(7,( 11, 12, 13, 18, 19, 24, 25)),
new wg_type'(7,( 11, 12, 13, 18, 23, 24, 25)),
new wg_type'(7,( 11, 12, 13, 18, 23, 24, 20)),
new wg_type'(5,( 11, 12,  8,  9, 10)),
new wg_type'(5,( 11, 12,  8,  9,  5)),
new wg_type'(6,( 11, 12,  8,  9, 14, 15)),
new wg_type'(9,( 11, 12,  8,  9, 14, 18, 23, 24, 25)),
new wg_type'(9,( 11, 12,  8,  9, 14, 18, 23, 24, 20)),
new wg_type'(7,( 11, 12,  8,  9, 14, 19, 20)),
new wg_type'(8,( 11, 12,  8,  9, 14, 19, 24, 25)),
new wg_type'(5,( 11, 12,  8,  4,  5)),
new wg_type'(6,( 11, 12, 17, 18, 19, 20)),
new wg_type'(6,( 11, 12, 17, 18, 19, 15)),
new wg_type'(7,( 11, 12, 17, 18, 19, 24, 25)),
new wg_type'(6,( 11, 12, 17, 18, 14, 15)),
new wg_type'(6,( 11, 12, 17, 18, 14, 10)),
new wg_type'(7,( 11, 12, 17, 18, 14,  9,  5)),
new wg_type'(7,( 11, 12, 17, 18, 23, 24, 25)),
new wg_type'(7,( 11, 12, 17, 18, 23, 24, 20)),
new wg_type'(7,( 11, 12, 17, 22, 23, 24, 25)),
new wg_type'(7,( 11, 12, 17, 22, 23, 24, 20)),
```

```
new wg_type'(7,( 11, 12, 17, 22, 23, 19, 20)),
new wg_type'(7,( 11, 12, 17, 22, 23, 19, 15)),
new wg_type'(8,( 11, 12, 17, 22, 23, 19, 14, 10)),
new wg_type'(9,( 11, 12, 17, 22, 23, 19, 14,  9,  5)),
new wg_type'(5,( 11,  7,  8,  9, 10)),
new wg_type'(5,( 11,  7,  8,  9,  5)),
new wg_type'(6,( 11,  7,  8,  9, 14, 15)),
new wg_type'(9,( 11,  7,  8,  9, 14, 18, 23, 24, 25)),
new wg_type'(9,( 11,  7,  8,  9, 14, 18, 23, 24, 20)),
new wg_type'(7,( 11,  7,  8,  9, 14, 19, 20)),
new wg_type'(8,( 11,  7,  8,  9, 14, 19, 24, 25)),
new wg_type'(5,( 11,  7,  8,  4,  5)),
new wg_type'(6,( 11,  7,  8, 13, 14, 15)),
new wg_type'(6,( 11,  7,  8, 13, 14, 10)),
new wg_type'(7,( 11,  7,  8, 13, 14, 19, 20)),
new wg_type'(8,( 11,  7,  8, 13, 14, 19, 24, 25)),
new wg_type'(9,( 11,  7,  8, 13, 17, 22, 23, 24, 25)),
new wg_type'(9,( 11,  7,  8, 13, 17, 22, 23, 24, 20)),
new wg_type'(9,( 11,  7,  8, 13, 17, 22, 23, 19, 20)),
new wg_type'(9,( 11,  7,  8, 13, 17, 22, 23, 19, 15)),
new wg_type'(7,( 11,  7,  8, 13, 18, 19, 20)),
new wg_type'(7,( 11,  7,  8, 13, 18, 19, 15)),
new wg_type'(8,( 11,  7,  8, 13, 18, 19, 24, 25)),
new wg_type'(8,( 11,  7,  8, 13, 18, 23, 24, 25)),
new wg_type'(8,( 11,  7,  8, 13, 18, 23, 24, 20)),
new wg_type'(5,( 11,  7,  3,  4,  5)),
new wg_type'(6,( 11,  7,  3,  4,  9, 10)),
new wg_type'(  11,( 11,  7,  3,  4,  9, 13, 17, 22, 23, 24, 25)),
new wg_type'(  11,( 11,  7,  3,  4,  9, 13, 17, 22, 23, 24, 20)),
new wg_type'(  11,( 11,  7,  3,  4,  9, 13, 17, 22, 23, 19, 20)),
new wg_type'(  11,( 11,  7,  3,  4,  9, 13, 17, 22, 23, 19, 15)),
new wg_type'(9,( 11,  7,  3,  4,  9, 13, 18, 19, 20)),
new wg_type'(9,( 11,  7,  3,  4,  9, 13, 18, 19, 15)),
new wg_type'(  10,( 11,  7,  3,  4,  9, 13, 18, 19, 24, 25)),
new wg_type'(  10,( 11,  7,  3,  4,  9, 13, 18, 23, 24, 25)),
new wg_type'(  10,( 11,  7,  3,  4,  9, 13, 18, 23, 24, 20)),
new wg_type'(7,( 11,  7,  3,  4,  9, 14, 15)),
new wg_type'(  10,( 11,  7,  3,  4,  9, 14, 18, 23, 24, 25)),
new wg_type'(  10,( 11,  7,  3,  4,  9, 14, 18, 23, 24, 20)),
new wg_type'(8,( 11,  7,  3,  4,  9, 14, 19, 20)),
new wg_type'(9,( 11,  7,  3,  4,  9, 14, 19, 24, 25)),
new wg_type'(5,( 16, 17, 18, 19, 20)),
new wg_type'(5,( 16, 17, 18, 19, 15)),
new wg_type'(6,( 16, 17, 18, 19, 24, 25)),
new wg_type'(5,( 16, 17, 18, 14, 15)),
new wg_type'(5,( 16, 17, 18, 14, 10)),
new wg_type'(6,( 16, 17, 18, 14,  9,  5)),
new wg_type'(6,( 16, 17, 18, 23, 24, 25)),
new wg_type'(6,( 16, 17, 18, 23, 24, 20)),
```

```
new wg_type'(5,( 16, 17, 13, 14, 15)),
new wg_type'(5,( 16, 17, 13, 14, 10)),
new wg_type'(6,( 16, 17, 13, 14, 19, 20)),
new wg_type'(7,( 16, 17, 13, 14, 19, 24, 25)),
new wg_type'(5,( 16, 17, 13,  9, 10)),
new wg_type'(5,( 16, 17, 13,  9,  5)),
new wg_type'(6,( 16, 17, 13,  8,  4,  5)),
new wg_type'(6,( 16, 17, 22, 23, 24, 25)),
new wg_type'(6,( 16, 17, 22, 23, 24, 20)),
new wg_type'(6,( 16, 17, 22, 23, 19, 20)),
new wg_type'(6,( 16, 17, 22, 23, 19, 15)),
new wg_type'(7,( 16, 17, 22, 23, 19, 14, 10)),
new wg_type'(8,( 16, 17, 22, 23, 19, 14,  9,  5)),
new wg_type'(5,( 16, 12, 13, 14, 15)),
new wg_type'(5,( 16, 12, 13, 14, 10)),
new wg_type'(6,( 16, 12, 13, 14, 19, 20)),
new wg_type'(7,( 16, 12, 13, 14, 19, 24, 25)),
new wg_type'(5,( 16, 12, 13,  9, 10)),
new wg_type'(5,( 16, 12, 13,  9,  5)),
new wg_type'(6,( 16, 12, 13, 18, 19, 20)),
new wg_type'(6,( 16, 12, 13, 18, 19, 15)),
new wg_type'(7,( 16, 12, 13, 18, 19, 24, 25)),
new wg_type'(7,( 16, 12, 13, 18, 23, 24, 25)),
new wg_type'(7,( 16, 12, 13, 18, 23, 24, 20)),
new wg_type'(5,( 16, 12,  8,  9, 10)),
new wg_type'(5,( 16, 12,  8,  9,  5)),
new wg_type'(6,( 16, 12,  8,  9, 14, 15)),
new wg_type'(9,( 16, 12,  8,  9, 14, 18, 23, 24, 25)),
new wg_type'(9,( 16, 12,  8,  9, 14, 18, 23, 24, 20)),
new wg_type'(7,( 16, 12,  8,  9, 14, 19, 20)),
new wg_type'(8,( 16, 12,  8,  9, 14, 19, 24, 25)),
new wg_type'(5,( 16, 12,  8,  4,  5)),
new wg_type'(6,( 16, 12,  7,  3,  4,  5)),
new wg_type'(7,( 16, 12,  7,  3,  4,  9, 10)),
new wg_type'(8,( 16, 12,  7,  3,  4,  9, 14, 15)),
new wg_type'(  11,( 16, 12,  7,  3,  4,  9, 14, 18, 23, 24, 25)),
new wg_type'(  11,( 16, 12,  7,  3,  4,  9, 14, 18, 23, 24, 20)),
new wg_type'(9,( 16, 12,  7,  3,  4,  9, 14, 19, 20)),
new wg_type'(  10,( 16, 12,  7,  3,  4,  9, 14, 19, 24, 25)),
new wg_type'(5,( 21, 22, 23, 24, 25)),
new wg_type'(5,( 21, 22, 23, 24, 20)),
new wg_type'(5,( 21, 22, 23, 19, 20)),
new wg_type'(5,( 21, 22, 23, 19, 15)),
new wg_type'(  11,( 21, 22, 23, 19, 14, 13, 12,  7,  3,  4,  5)),
new wg_type'(9,( 21, 22, 23, 19, 14, 13,  8,  4,  5)),
new wg_type'(6,( 21, 22, 23, 19, 14, 10)),
new wg_type'(7,( 21, 22, 23, 19, 14,  9,  5)),
new wg_type'(5,( 21, 22, 18, 19, 20)),
new wg_type'(5,( 21, 22, 18, 19, 15)),
```

```
new wg_type'(6,( 21, 22, 18, 19, 24, 25)),
new wg_type'(5,( 21, 22, 18, 14, 15)),
new wg_type'(5,( 21, 22, 18, 14, 10)),
new wg_type'(6,( 21, 22, 18, 14,  9,  5)),
new wg_type'(9,( 21, 22, 18, 13, 12,  7,  3,  4,  5)),
new wg_type'(6,( 21, 22, 18, 13,  9, 10)),
new wg_type'(6,( 21, 22, 18, 13,  9,  5)),
new wg_type'(7,( 21, 22, 18, 13,  8,  4,  5)),
new wg_type'(5,( 21, 17, 18, 19, 20)),
new wg_type'(5,( 21, 17, 18, 19, 15)),
new wg_type'(6,( 21, 17, 18, 19, 24, 25)),
new wg_type'(5,( 21, 17, 18, 14, 15)),
new wg_type'(5,( 21, 17, 18, 14, 10)),
new wg_type'(6,( 21, 17, 18, 14,  9,  5)),
new wg_type'(6,( 21, 17, 18, 23, 24, 25)),
new wg_type'(6,( 21, 17, 18, 23, 24, 20)),
new wg_type'(5,( 21, 17, 13, 14, 15)),
new wg_type'(5,( 21, 17, 13, 14, 10)),
new wg_type'(6,( 21, 17, 13, 14, 19, 20)),
new wg_type'(7,( 21, 17, 13, 14, 19, 24, 25)),
new wg_type'(5,( 21, 17, 13,  9, 10)),
new wg_type'(5,( 21, 17, 13,  9,  5)),
new wg_type'(6,( 21, 17, 13,  8,  4,  5)),
new wg_type'(6,( 21, 17, 12,  8,  9, 10)),
new wg_type'(6,( 21, 17, 12,  8,  9,  5)),
new wg_type'(7,( 21, 17, 12,  8,  9, 14, 15)),
new wg_type'(8,( 21, 17, 12,  8,  9, 14, 19, 20)),
new wg_type'(9,( 21, 17, 12,  8,  9, 14, 19, 24, 25)),
new wg_type'(6,( 21, 17, 12,  8,  4,  5)),
new wg_type'(7,( 21, 17, 12,  7,  3,  4,  5)),
new wg_type'(8,( 21, 17, 12,  7,  3,  4,  9, 10)),
new wg_type'(9,( 21, 17, 12,  7,  3,  4,  9, 14, 15)),
new wg_type'(  10,( 21, 17, 12,  7,  3,  4,  9, 14, 19, 20)),
new wg_type'(  11,( 21, 17, 12,  7,  3,  4,  9, 14, 19, 24, 25)));

wgb : constant wg_table(1..365) :=
(
new wg_type'(  11,(  1,  6,  7,  8,  9, 10, 15, 19, 18, 17, 21)),
new wg_type'(  10,(  1,  6,  7,  8,  9, 10, 15, 19, 18, 22)),
new wg_type'(9,(  1,  6,  7,  8,  9, 10, 15, 19, 23)),
new wg_type'(9,(  1,  6,  7,  8,  9, 10, 15, 19, 24)),
new wg_type'(9,(  1,  6,  7,  8,  9, 10, 15, 20, 24)),
new wg_type'(9,(  1,  6,  7,  8,  9, 10, 15, 20, 25)),
new wg_type'(9,(  1,  6,  7,  8,  9, 14, 15, 20, 24)),
new wg_type'(9,(  1,  6,  7,  8,  9, 14, 15, 20, 25)),
new wg_type'(9,(  1,  6,  7,  8,  9, 14, 18, 17, 21)),
new wg_type'(8,(  1,  6,  7,  8,  9, 14, 18, 22)),
new wg_type'(8,(  1,  6,  7,  8,  9, 14, 18, 23)),
new wg_type'(9,(  1,  6,  7,  8,  9, 14, 19, 20, 25)),
```

```
new wg_type'(8,(  1,  6,  7,  8,  9, 14, 19, 23)),
new wg_type'(8,(  1,  6,  7,  8,  9, 14, 19, 24)),
new wg_type'(9,(  1,  6,  7,  8, 13, 14, 15, 20, 24)),
new wg_type'(9,(  1,  6,  7,  8, 13, 14, 15, 20, 25)),
new wg_type'(9,(  1,  6,  7,  8, 13, 14, 19, 20, 25)),
new wg_type'(8,(  1,  6,  7,  8, 13, 14, 19, 23)),
new wg_type'(8,(  1,  6,  7,  8, 13, 14, 19, 24)),
new wg_type'(7,(  1,  6,  7,  8, 13, 17, 21)),
new wg_type'(7,(  1,  6,  7,  8, 13, 17, 22)),
new wg_type'(9,(  1,  6,  7,  8, 13, 18, 19, 20, 25)),
new wg_type'(8,(  1,  6,  7,  8, 13, 18, 19, 24)),
new wg_type'(7,(  1,  6,  7,  8, 13, 18, 22)),
new wg_type'(7,(  1,  6,  7,  8, 13, 18, 23)),
new wg_type'(9,(  1,  6,  7, 12, 13, 14, 15, 20, 24)),
new wg_type'(9,(  1,  6,  7, 12, 13, 14, 15, 20, 25)),
new wg_type'(9,(  1,  6,  7, 12, 13, 14, 19, 20, 25)),
new wg_type'(8,(  1,  6,  7, 12, 13, 14, 19, 23)),
new wg_type'(8,(  1,  6,  7, 12, 13, 14, 19, 24)),
new wg_type'(  10,(  1,  6,  7, 12, 13,  9, 10, 15, 19, 23)),
new wg_type'(  10,(  1,  6,  7, 12, 13,  9, 10, 15, 19, 24)),
new wg_type'(  10,(  1,  6,  7, 12, 13,  9, 10, 15, 20, 24)),
new wg_type'(  10,(  1,  6,  7, 12, 13,  9, 10, 15, 20, 25)),
new wg_type'(9,(  1,  6,  7, 12, 13, 18, 19, 20, 25)),
new wg_type'(8,(  1,  6,  7, 12, 13, 18, 19, 24)),
new wg_type'(7,(  1,  6,  7, 12, 13, 18, 22)),
new wg_type'(7,(  1,  6,  7, 12, 13, 18, 23)),
new wg_type'(6,(  1,  6,  7, 12, 16, 21)),
new wg_type'(9,(  1,  6,  7, 12, 17, 18, 19, 20, 25)),
new wg_type'(8,(  1,  6,  7, 12, 17, 18, 19, 24)),
new wg_type'(  10,(  1,  6,  7, 12, 17, 18, 14, 15, 20, 24)),
new wg_type'(  10,(  1,  6,  7, 12, 17, 18, 14, 15, 20, 25)),
new wg_type'(7,(  1,  6,  7, 12, 17, 18, 23)),
new wg_type'(6,(  1,  6,  7, 12, 17, 21)),
new wg_type'(6,(  1,  6,  7, 12, 17, 22)),
new wg_type'(9,(  1,  6, 11, 12, 13, 14, 15, 20, 24)),
new wg_type'(9,(  1,  6, 11, 12, 13, 14, 15, 20, 25)),
new wg_type'(9,(  1,  6, 11, 12, 13, 14, 19, 20, 25)),
new wg_type'(8,(  1,  6, 11, 12, 13, 14, 19, 23)),
new wg_type'(8,(  1,  6, 11, 12, 13, 14, 19, 24)),
new wg_type'(  10,(  1,  6, 11, 12, 13,  9, 10, 15, 19, 23)),
new wg_type'(  10,(  1,  6, 11, 12, 13,  9, 10, 15, 19, 24)),
new wg_type'(  10,(  1,  6, 11, 12, 13,  9, 10, 15, 20, 24)),
new wg_type'(  10,(  1,  6, 11, 12, 13,  9, 10, 15, 20, 25)),
new wg_type'(9,(  1,  6, 11, 12, 13, 18, 19, 20, 25)),
new wg_type'(8,(  1,  6, 11, 12, 13, 18, 19, 24)),
new wg_type'(7,(  1,  6, 11, 12, 13, 18, 22)),
new wg_type'(7,(  1,  6, 11, 12, 13, 18, 23)),
new wg_type'(  11,(  1,  6, 11, 12,  8,  9, 10, 15, 19, 18, 22)),
new wg_type'(  10,(  1,  6, 11, 12,  8,  9, 10, 15, 19, 23)),
```

```
new wg_type'(  10,(  1,  6, 11, 12,  8,  9, 10, 15, 19, 24)),
new wg_type'(  10,(  1,  6, 11, 12,  8,  9, 10, 15, 20, 24)),
new wg_type'(  10,(  1,  6, 11, 12,  8,  9, 10, 15, 20, 25)),
new wg_type'(  10,(  1,  6, 11, 12,  8,  9, 14, 15, 20, 24)),
new wg_type'(  10,(  1,  6, 11, 12,  8,  9, 14, 15, 20, 25)),
new wg_type'(9,(  1,  6, 11, 12,  8,  9, 14, 18, 22)),
new wg_type'(9,(  1,  6, 11, 12,  8,  9, 14, 18, 23)),
new wg_type'(  10,(  1,  6, 11, 12,  8,  9, 14, 19, 20, 25)),
new wg_type'(9,(  1,  6, 11, 12,  8,  9, 14, 19, 23)),
new wg_type'(9,(  1,  6, 11, 12,  8,  9, 14, 19, 24)),
new wg_type'(9,(  1,  6, 11, 12, 17, 18, 19, 20, 25)),
new wg_type'(8,(  1,  6, 11, 12, 17, 18, 19, 24)),
new wg_type'(  10,(  1,  6, 11, 12, 17, 18, 14, 15, 20, 24)),
new wg_type'(  10,(  1,  6, 11, 12, 17, 18, 14, 15, 20, 25)),
new wg_type'(7,(  1,  6, 11, 12, 17, 18, 23)),
new wg_type'(6,(  1,  6, 11, 12, 17, 21)),
new wg_type'(6,(  1,  6, 11, 12, 17, 22)),
new wg_type'(9,(  1,  6, 11, 16, 17, 18, 19, 20, 25)),
new wg_type'(8,(  1,  6, 11, 16, 17, 18, 19, 24)),
new wg_type'(  10,(  1,  6, 11, 16, 17, 18, 14, 15, 20, 24)),
new wg_type'(  10,(  1,  6, 11, 16, 17, 18, 14, 15, 20, 25)),
new wg_type'(7,(  1,  6, 11, 16, 17, 18, 23)),
new wg_type'(  10,(  1,  6, 11, 16, 17, 13, 14, 15, 20, 24)),
new wg_type'(  10,(  1,  6, 11, 16, 17, 13, 14, 15, 20, 25)),
new wg_type'(  10,(  1,  6, 11, 16, 17, 13, 14, 19, 20, 25)),
new wg_type'(9,(  1,  6, 11, 16, 17, 13, 14, 19, 23)),
new wg_type'(9,(  1,  6, 11, 16, 17, 13, 14, 19, 24)),
new wg_type'(  11,(  1,  6, 11, 16, 17, 13,  9, 10, 15, 19, 23)),
new wg_type'(  11,(  1,  6, 11, 16, 17, 13,  9, 10, 15, 19, 24)),
new wg_type'(  11,(  1,  6, 11, 16, 17, 13,  9, 10, 15, 20, 24)),
new wg_type'(  11,(  1,  6, 11, 16, 17, 13,  9, 10, 15, 20, 25)),
new wg_type'(6,(  1,  6, 11, 16, 17, 22)),
new wg_type'(5,(  1,  6, 11, 16, 21)),
new wg_type'(9,(  2,  6, 11, 12, 13, 14, 15, 20, 24)),
new wg_type'(9,(  2,  6, 11, 12, 13, 14, 15, 20, 25)),
new wg_type'(9,(  2,  6, 11, 12, 13, 14, 19, 20, 25)),
new wg_type'(8,(  2,  6, 11, 12, 13, 14, 19, 23)),
new wg_type'(8,(  2,  6, 11, 12, 13, 14, 19, 24)),
new wg_type'(  10,(  2,  6, 11, 12, 13,  9, 10, 15, 19, 23)),
new wg_type'(  10,(  2,  6, 11, 12, 13,  9, 10, 15, 19, 24)),
new wg_type'(  10,(  2,  6, 11, 12, 13,  9, 10, 15, 20, 24)),
new wg_type'(  10,(  2,  6, 11, 12, 13,  9, 10, 15, 20, 25)),
new wg_type'(9,(  2,  6, 11, 12, 13, 18, 19, 20, 25)),
new wg_type'(8,(  2,  6, 11, 12, 13, 18, 19, 24)),
new wg_type'(7,(  2,  6, 11, 12, 13, 18, 22)),
new wg_type'(7,(  2,  6, 11, 12, 13, 18, 23)),
new wg_type'(  11,(  2,  6, 11, 12,  8,  9, 10, 15, 19, 18, 22)),
new wg_type'(  10,(  2,  6, 11, 12,  8,  9, 10, 15, 19, 23)),
new wg_type'(  10,(  2,  6, 11, 12,  8,  9, 10, 15, 19, 24)),
```

```
new wg_type'(  10,(  2,  6, 11, 12,  8,  9, 10, 15, 20, 24)),
new wg_type'(  10,(  2,  6, 11, 12,  8,  9, 10, 15, 20, 25)),
new wg_type'(  10,(  2,  6, 11, 12,  8,  9, 14, 15, 20, 24)),
new wg_type'(  10,(  2,  6, 11, 12,  8,  9, 14, 15, 20, 25)),
new wg_type'(9,(  2,  6, 11, 12,  8,  9, 14, 18, 22)),
new wg_type'(9,(  2,  6, 11, 12,  8,  9, 14, 18, 23)),
new wg_type'(  10,(  2,  6, 11, 12,  8,  9, 14, 19, 20, 25)),
new wg_type'(9,(  2,  6, 11, 12,  8,  9, 14, 19, 23)),
new wg_type'(9,(  2,  6, 11, 12,  8,  9, 14, 19, 24)),
new wg_type'(9,(  2,  6, 11, 12, 17, 18, 19, 20, 25)),
new wg_type'(8,(  2,  6, 11, 12, 17, 18, 19, 24)),
new wg_type'(  10,(  2,  6, 11, 12, 17, 18, 14, 15, 20, 24)),
new wg_type'(  10,(  2,  6, 11, 12, 17, 18, 14, 15, 20, 25)),
new wg_type'(7,(  2,  6, 11, 12, 17, 18, 23)),
new wg_type'(6,(  2,  6, 11, 12, 17, 21)),
new wg_type'(6,(  2,  6, 11, 12, 17, 22)),
new wg_type'(9,(  2,  6, 11, 16, 17, 18, 19, 20, 25)),
new wg_type'(8,(  2,  6, 11, 16, 17, 18, 19, 24)),
new wg_type'(  10,(  2,  6, 11, 16, 17, 18, 14, 15, 20, 24)),
new wg_type'(  10,(  2,  6, 11, 16, 17, 18, 14, 15, 20, 25)),
new wg_type'(7,(  2,  6, 11, 16, 17, 18, 23)),
new wg_type'(  10,(  2,  6, 11, 16, 17, 13, 14, 15, 20, 24)),
new wg_type'(  10,(  2,  6, 11, 16, 17, 13, 14, 15, 20, 25)),
new wg_type'(  10,(  2,  6, 11, 16, 17, 13, 14, 19, 20, 25)),
new wg_type'(9,(  2,  6, 11, 16, 17, 13, 14, 19, 23)),
new wg_type'(9,(  2,  6, 11, 16, 17, 13, 14, 19, 24)),
new wg_type'(  11,(  2,  6, 11, 16, 17, 13,  9, 10, 15, 19, 23)),
new wg_type'(  11,(  2,  6, 11, 16, 17, 13,  9, 10, 15, 19, 24)),
new wg_type'(  11,(  2,  6, 11, 16, 17, 13,  9, 10, 15, 20, 24)),
new wg_type'(  11,(  2,  6, 11, 16, 17, 13,  9, 10, 15, 20, 25)),
new wg_type'(6,(  2,  6, 11, 16, 17, 22)),
new wg_type'(5,(  2,  6, 11, 16, 21)),
new wg_type'(  10,(  2,  7,  8,  9, 10, 15, 19, 18, 17, 21)),
new wg_type'(9,(  2,  7,  8,  9, 10, 15, 19, 18, 22)),
new wg_type'(8,(  2,  7,  8,  9, 10, 15, 19, 23)),
new wg_type'(8,(  2,  7,  8,  9, 10, 15, 19, 24)),
new wg_type'(8,(  2,  7,  8,  9, 10, 15, 20, 24)),
new wg_type'(8,(  2,  7,  8,  9, 10, 15, 20, 25)),
new wg_type'(8,(  2,  7,  8,  9, 14, 15, 20, 24)),
new wg_type'(8,(  2,  7,  8,  9, 14, 15, 20, 25)),
new wg_type'(8,(  2,  7,  8,  9, 14, 18, 17, 21)),
new wg_type'(7,(  2,  7,  8,  9, 14, 18, 22)),
new wg_type'(7,(  2,  7,  8,  9, 14, 18, 23)),
new wg_type'(8,(  2,  7,  8,  9, 14, 19, 20, 25)),
new wg_type'(7,(  2,  7,  8,  9, 14, 19, 23)),
new wg_type'(7,(  2,  7,  8,  9, 14, 19, 24)),
new wg_type'(8,(  2,  7,  8, 13, 14, 15, 20, 24)),
new wg_type'(8,(  2,  7,  8, 13, 14, 15, 20, 25)),
new wg_type'(8,(  2,  7,  8, 13, 14, 19, 20, 25)),
```

```
new wg_type'(7,(  2,  7,  8, 13, 14, 19, 23)),
new wg_type'(7,(  2,  7,  8, 13, 14, 19, 24)),
new wg_type'(6,(  2,  7,  8, 13, 17, 21)),
new wg_type'(6,(  2,  7,  8, 13, 17, 22)),
new wg_type'(8,(  2,  7,  8, 13, 18, 19, 20, 25)),
new wg_type'(7,(  2,  7,  8, 13, 18, 19, 24)),
new wg_type'(6,(  2,  7,  8, 13, 18, 22)),
new wg_type'(6,(  2,  7,  8, 13, 18, 23)),
new wg_type'(9,(  2,  7, 11, 16, 17, 18, 19, 20, 25)),
new wg_type'(8,(  2,  7, 11, 16, 17, 18, 19, 24)),
new wg_type'(  10,(  2,  7, 11, 16, 17, 18, 14, 15, 20, 24)),
new wg_type'(  10,(  2,  7, 11, 16, 17, 18, 14, 15, 20, 25)),
new wg_type'(7,(  2,  7, 11, 16, 17, 18, 23)),
new wg_type'(  10,(  2,  7, 11, 16, 17, 13, 14, 15, 20, 24)),
new wg_type'(  10,(  2,  7, 11, 16, 17, 13, 14, 15, 20, 25)),
new wg_type'(  10,(  2,  7, 11, 16, 17, 13, 14, 19, 20, 25)),
new wg_type'(9,(  2,  7, 11, 16, 17, 13, 14, 19, 23)),
new wg_type'(9,(  2,  7, 11, 16, 17, 13, 14, 19, 24)),
new wg_type'(  11,(  2,  7, 11, 16, 17, 13,  9, 10, 15, 19, 23)),
new wg_type'(  11,(  2,  7, 11, 16, 17, 13,  9, 10, 15, 19, 24)),
new wg_type'(  11,(  2,  7, 11, 16, 17, 13,  9, 10, 15, 20, 24)),
new wg_type'(  11,(  2,  7, 11, 16, 17, 13,  9, 10, 15, 20, 25)),
new wg_type'(6,(  2,  7, 11, 16, 17, 22)),
new wg_type'(5,(  2,  7, 11, 16, 21)),
new wg_type'(8,(  2,  7, 12, 13, 14, 15, 20, 24)),
new wg_type'(8,(  2,  7, 12, 13, 14, 15, 20, 25)),
new wg_type'(8,(  2,  7, 12, 13, 14, 19, 20, 25)),
new wg_type'(7,(  2,  7, 12, 13, 14, 19, 23)),
new wg_type'(7,(  2,  7, 12, 13, 14, 19, 24)),
new wg_type'(9,(  2,  7, 12, 13,  9, 10, 15, 19, 23)),
new wg_type'(9,(  2,  7, 12, 13,  9, 10, 15, 19, 24)),
new wg_type'(9,(  2,  7, 12, 13,  9, 10, 15, 20, 24)),
new wg_type'(9,(  2,  7, 12, 13,  9, 10, 15, 20, 25)),
new wg_type'(8,(  2,  7, 12, 13, 18, 19, 20, 25)),
new wg_type'(7,(  2,  7, 12, 13, 18, 19, 24)),
new wg_type'(6,(  2,  7, 12, 13, 18, 22)),
new wg_type'(6,(  2,  7, 12, 13, 18, 23)),
new wg_type'(5,(  2,  7, 12, 16, 21)),
new wg_type'(8,(  2,  7, 12, 17, 18, 19, 20, 25)),
new wg_type'(7,(  2,  7, 12, 17, 18, 19, 24)),
new wg_type'(9,(  2,  7, 12, 17, 18, 14, 15, 20, 24)),
new wg_type'(9,(  2,  7, 12, 17, 18, 14, 15, 20, 25)),
new wg_type'(6,(  2,  7, 12, 17, 18, 23)),
new wg_type'(5,(  2,  7, 12, 17, 21)),
new wg_type'(5,(  2,  7, 12, 17, 22)),
new wg_type'(9,(  3,  7, 11, 16, 17, 18, 19, 20, 25)),
new wg_type'(8,(  3,  7, 11, 16, 17, 18, 19, 24)),
new wg_type'(  10,(  3,  7, 11, 16, 17, 18, 14, 15, 20, 24)),
new wg_type'(  10,(  3,  7, 11, 16, 17, 18, 14, 15, 20, 25)),
```

```
new wg_type'(7,(  3,  7, 11, 16, 17, 18, 23)),
new wg_type'(  10,(  3,  7, 11, 16, 17, 13, 14, 15, 20, 24)),
new wg_type'(  10,(  3,  7, 11, 16, 17, 13, 14, 15, 20, 25)),
new wg_type'(  10,(  3,  7, 11, 16, 17, 13, 14, 19, 20, 25)),
new wg_type'(9,(  3,  7, 11, 16, 17, 13, 14, 19, 23)),
new wg_type'(9,(  3,  7, 11, 16, 17, 13, 14, 19, 24)),
new wg_type'(  11,(  3,  7, 11, 16, 17, 13,  9, 10, 15, 19, 23)),
new wg_type'(  11,(  3,  7, 11, 16, 17, 13,  9, 10, 15, 19, 24)),
new wg_type'(  11,(  3,  7, 11, 16, 17, 13,  9, 10, 15, 20, 24)),
new wg_type'(  11,(  3,  7, 11, 16, 17, 13,  9, 10, 15, 20, 25)),
new wg_type'(6,(  3,  7, 11, 16, 17, 22)),
new wg_type'(5,(  3,  7, 11, 16, 21)),
new wg_type'(8,(  3,  7, 12, 13, 14, 15, 20, 24)),
new wg_type'(8,(  3,  7, 12, 13, 14, 15, 20, 25)),
new wg_type'(8,(  3,  7, 12, 13, 14, 19, 20, 25)),
new wg_type'(7,(  3,  7, 12, 13, 14, 19, 23)),
new wg_type'(7,(  3,  7, 12, 13, 14, 19, 24)),
new wg_type'(9,(  3,  7, 12, 13,  9, 10, 15, 19, 23)),
new wg_type'(9,(  3,  7, 12, 13,  9, 10, 15, 19, 24)),
new wg_type'(9,(  3,  7, 12, 13,  9, 10, 15, 20, 24)),
new wg_type'(9,(  3,  7, 12, 13,  9, 10, 15, 20, 25)),
new wg_type'(8,(  3,  7, 12, 13, 18, 19, 20, 25)),
new wg_type'(7,(  3,  7, 12, 13, 18, 19, 24)),
new wg_type'(6,(  3,  7, 12, 13, 18, 22)),
new wg_type'(6,(  3,  7, 12, 13, 18, 23)),
new wg_type'(5,(  3,  7, 12, 16, 21)),
new wg_type'(8,(  3,  7, 12, 17, 18, 19, 20, 25)),
new wg_type'(7,(  3,  7, 12, 17, 18, 19, 24)),
new wg_type'(9,(  3,  7, 12, 17, 18, 14, 15, 20, 24)),
new wg_type'(9,(  3,  7, 12, 17, 18, 14, 15, 20, 25)),
new wg_type'(6,(  3,  7, 12, 17, 18, 23)),
new wg_type'(5,(  3,  7, 12, 17, 21)),
new wg_type'(5,(  3,  7, 12, 17, 22)),
new wg_type'(9,(  3,  8,  9, 10, 15, 19, 18, 17, 21)),
new wg_type'(8,(  3,  8,  9, 10, 15, 19, 18, 22)),
new wg_type'(7,(  3,  8,  9, 10, 15, 19, 23)),
new wg_type'(7,(  3,  8,  9, 10, 15, 19, 24)),
new wg_type'(7,(  3,  8,  9, 10, 15, 20, 24)),
new wg_type'(7,(  3,  8,  9, 10, 15, 20, 25)),
new wg_type'(7,(  3,  8,  9, 14, 15, 20, 24)),
new wg_type'(7,(  3,  8,  9, 14, 15, 20, 25)),
new wg_type'(7,(  3,  8,  9, 14, 18, 17, 21)),
new wg_type'(6,(  3,  8,  9, 14, 18, 22)),
new wg_type'(6,(  3,  8,  9, 14, 18, 23)),
new wg_type'(7,(  3,  8,  9, 14, 19, 20, 25)),
new wg_type'(6,(  3,  8,  9, 14, 19, 23)),
new wg_type'(6,(  3,  8,  9, 14, 19, 24)),
new wg_type'(5,(  3,  8, 12, 16, 21)),
new wg_type'(8,(  3,  8, 12, 17, 18, 19, 20, 25)),
```

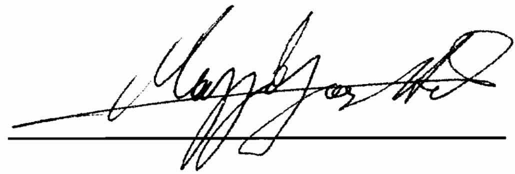
```
new wg_type'(7,(  3,  8, 12, 17, 18, 19, 24)),
new wg_type'(9,(  3,  8, 12, 17, 18, 14, 15, 20, 24)),
new wg_type'(9,(  3,  8, 12, 17, 18, 14, 15, 20, 25)),
new wg_type'(6,(  3,  8, 12, 17, 18, 23)),
new wg_type'(5,(  3,  8, 12, 17, 21)),
new wg_type'(5,(  3,  8, 12, 17, 22)),
new wg_type'(7,(  3,  8, 13, 14, 15, 20, 24)),
new wg_type'(7,(  3,  8, 13, 14, 15, 20, 25)),
new wg_type'(7,(  3,  8, 13, 14, 19, 20, 25)),
new wg_type'(6,(  3,  8, 13, 14, 19, 23)),
new wg_type'(6,(  3,  8, 13, 14, 19, 24)),
new wg_type'(5,(  3,  8, 13, 17, 21)),
new wg_type'(5,(  3,  8, 13, 17, 22)),
new wg_type'(7,(  3,  8, 13, 18, 19, 20, 25)),
new wg_type'(6,(  3,  8, 13, 18, 19, 24)),
new wg_type'(5,(  3,  8, 13, 18, 22)),
new wg_type'(5,(  3,  8, 13, 18, 23)),
new wg_type'(  10,(  4,  8,  7, 11, 16, 17, 18, 19, 20, 25)),
new wg_type'(9,(  4,  8,  7, 11, 16, 17, 18, 19, 24)),
new wg_type'(  11,(  4,  8,  7, 11, 16, 17, 18, 14, 15, 20, 24)),
new wg_type'(  11,(  4,  8,  7, 11, 16, 17, 18, 14, 15, 20, 25)),
new wg_type'(8,(  4,  8,  7, 11, 16, 17, 18, 23)),
new wg_type'(7,(  4,  8,  7, 11, 16, 17, 22)),
new wg_type'(6,(  4,  8,  7, 11, 16, 21)),
new wg_type'(5,(  4,  8, 12, 16, 21)),
new wg_type'(8,(  4,  8, 12, 17, 18, 19, 20, 25)),
new wg_type'(7,(  4,  8, 12, 17, 18, 19, 24)),
new wg_type'(9,(  4,  8, 12, 17, 18, 14, 15, 20, 24)),
new wg_type'(9,(  4,  8, 12, 17, 18, 14, 15, 20, 25)),
new wg_type'(6,(  4,  8, 12, 17, 18, 23)),
new wg_type'(5,(  4,  8, 12, 17, 21)),
new wg_type'(5,(  4,  8, 12, 17, 22)),
new wg_type'(7,(  4,  8, 13, 14, 15, 20, 24)),
new wg_type'(7,(  4,  8, 13, 14, 15, 20, 25)),
new wg_type'(7,(  4,  8, 13, 14, 19, 20, 25)),
new wg_type'(6,(  4,  8, 13, 14, 19, 23)),
new wg_type'(6,(  4,  8, 13, 14, 19, 24)),
new wg_type'(5,(  4,  8, 13, 17, 21)),
new wg_type'(5,(  4,  8, 13, 17, 22)),
new wg_type'(7,(  4,  8, 13, 18, 19, 20, 25)),
new wg_type'(6,(  4,  8, 13, 18, 19, 24)),
new wg_type'(5,(  4,  8, 13, 18, 22)),
new wg_type'(5,(  4,  8, 13, 18, 23)),
new wg_type'(8,(  4,  9, 10, 15, 19, 18, 17, 21)),
new wg_type'(7,(  4,  9, 10, 15, 19, 18, 22)),
new wg_type'(6,(  4,  9, 10, 15, 19, 23)),
new wg_type'(6,(  4,  9, 10, 15, 19, 24)),
new wg_type'(6,(  4,  9, 10, 15, 20, 24)),
new wg_type'(6,(  4,  9, 10, 15, 20, 25)),
```

```
new wg_type'(6,(  4,  9, 13, 12, 16, 21)),
new wg_type'(5,(  4,  9, 13, 17, 21)),
new wg_type'(5,(  4,  9, 13, 17, 22)),
new wg_type'(7,(  4,  9, 13, 18, 19, 20, 25)),
new wg_type'(6,(  4,  9, 13, 18, 19, 24)),
new wg_type'(5,(  4,  9, 13, 18, 22)),
new wg_type'(5,(  4,  9, 13, 18, 23)),
new wg_type'(6,(  4,  9, 14, 15, 20, 24)),
new wg_type'(6,(  4,  9, 14, 15, 20, 25)),
new wg_type'(6,(  4,  9, 14, 18, 17, 21)),
new wg_type'(5,(  4,  9, 14, 18, 22)),
new wg_type'(5,(  4,  9, 14, 18, 23)),
new wg_type'(6,(  4,  9, 14, 19, 20, 25)),
new wg_type'(5,(  4,  9, 14, 19, 23)),
new wg_type'(5,(  4,  9, 14, 19, 24)),
new wg_type'(  11,(  5,  9,  8,  7, 11, 16, 17, 18, 19, 20, 25)),
new wg_type'(  10,(  5,  9,  8,  7, 11, 16, 17, 18, 19, 24)),
new wg_type'(9,(  5,  9,  8,  7, 11, 16, 17, 18, 23)),
new wg_type'(8,(  5,  9,  8,  7, 11, 16, 17, 22)),
new wg_type'(7,(  5,  9,  8,  7, 11, 16, 21)),
new wg_type'(6,(  5,  9,  8, 12, 16, 21)),
new wg_type'(9,(  5,  9,  8, 12, 17, 18, 19, 20, 25)),
new wg_type'(8,(  5,  9,  8, 12, 17, 18, 19, 24)),
new wg_type'(7,(  5,  9,  8, 12, 17, 18, 23)),
new wg_type'(6,(  5,  9,  8, 12, 17, 21)),
new wg_type'(6,(  5,  9,  8, 12, 17, 22)),
new wg_type'(6,(  5,  9, 13, 12, 16, 21)),
new wg_type'(5,(  5,  9, 13, 17, 21)),
new wg_type'(5,(  5,  9, 13, 17, 22)),
new wg_type'(7,(  5,  9, 13, 18, 19, 20, 25)),
new wg_type'(6,(  5,  9, 13, 18, 19, 24)),
new wg_type'(5,(  5,  9, 13, 18, 22)),
new wg_type'(5,(  5,  9, 13, 18, 23)),
new wg_type'(6,(  5,  9, 14, 15, 20, 24)),
new wg_type'(6,(  5,  9, 14, 15, 20, 25)),
new wg_type'(6,(  5,  9, 14, 18, 17, 21)),
new wg_type'(5,(  5,  9, 14, 18, 22)),
new wg_type'(5,(  5,  9, 14, 18, 23)),
new wg_type'(6,(  5,  9, 14, 19, 20, 25)),
new wg_type'(5,(  5,  9, 14, 19, 23)),
new wg_type'(5,(  5,  9, 14, 19, 24)),
new wg_type'(7,(  5, 10, 14, 13, 12, 16, 21)),
new wg_type'(6,(  5, 10, 14, 13, 17, 21)),
new wg_type'(6,(  5, 10, 14, 13, 17, 22)),
new wg_type'(9,(  5, 10, 14, 13,  8,  7, 11, 16, 21)),
new wg_type'(6,(  5, 10, 14, 18, 17, 21)),
new wg_type'(5,(  5, 10, 14, 18, 22)),
new wg_type'(5,(  5, 10, 14, 18, 23)),
new wg_type'(6,(  5, 10, 14, 19, 20, 25)),
```

```
new wg_type'(5,(  5, 10, 14, 19, 23)),
new wg_type'(5,(  5, 10, 14, 19, 24)),
new wg_type'(7,(  5, 10, 15, 19, 18, 17, 21)),
new wg_type'(6,(  5, 10, 15, 19, 18, 22)),
new wg_type'(9,(  5, 10, 15, 19, 18, 13, 12, 16, 21)),
new wg_type'(  11,(  5, 10, 15, 19, 18, 13,  8,  7, 11, 16, 21)),
new wg_type'(5,(  5, 10, 15, 19, 23)),
new wg_type'(5,(  5, 10, 15, 19, 24)),
new wg_type'(5,(  5, 10, 15, 20, 24)),
new wg_type'(5,(  5, 10, 15, 20, 25)));

end board ;
```

VITA

A. Mangolas was born in Athens, Greece, on March 6, 1965. He received his Bachelor of Science degree in Statistics from the University of Pireaus, Greece, in 1987. He has worked as programmer, analyst, and database designer for various companies in Greece.



Athanassios A. Mangolas