The Multigame Reference Manual

John W. Romein, Henri E. Bal, Dick Grune
{*john,bal,dick*}*@cs.vu.nl*

Faculty of Sciences
Dept. of Mathematics and Computer Science
Vrije Univeristeit
Amsterdam, The Netherlands

August 29, 2000

Abstract

This document describes Multigame, a language for expressing the rules of one-person and two-person board games, such as *chess*, *checkers*, and the *15-puzzle*. The programmer writes the rules of a game in a Multigame program. From this description, the Multigame compiler generates a (parallel) program that can play the game. The document gives both an informal introduction to the language and a formal definition of the grammar. The document also describes how a Multigame playing program communicates moves in a nearly game-independent way.

To improve the quality of the game-playing program, the programmer can provide an evaluation function with game-dependent heuristics. The evaluation function should be written in C; we describe the interface in this document as well.

1 Introduction

Multigame is an application-domain-oriented, high-level language for describing one-person and two-person board games. A Multigame program describes the legal moves of a game in a formal way. The Multigame front-end compiler uses this description to generate a (parallel) program that can play the game. The language offers the programmer a simple programming model. Moreover, the language hides parallel programming issues such as communication, synchronization, work and data distribution, and deadlock prevention from the programmer.

The class of problems that can be expressed in Multigame is restricted. The restrictions are the following:

- The language is designed for board games only. Concepts like *boards*, *pieces*, *fields*, *players*, and *moves* are embedded in the language. Card games (like *bridge*) and computer games (like *Quake*) cannot be expressed, since they are not based on these concepts.

- The game must be either a one-player game (e.g., the *15-puzzle*) or a two-player game (e.g., *chess*, *checkers*, *Othello*, *tic-tac-toe*).[1] The rules for both players in a two-person game need not be the same.
- The board consists of a rectangular, two-dimensional grid of fields. Each field holds at most one piece. Not all fields have to be used (this is useful for games like *checkers*, *nine men's morris*, and *Pegged*). The programmer can unfold a three-dimensional "board" (as used with *Rubik's cube*) to a two-dimensional representation.
- Perfect information is required. This excludes games like *Stratego* where one cannot see the identity of the opponent's pieces.
- Finally, games that depend on chance or probability are excluded. Therefore, it is not possible to play *backgammon* and *Risk*, because they require dice.

The Multigame compiler accepts a Multigame program and generates an ANSI-C program, that can play the game against a human opponent or against another computer. The generated program consists of four major parts:

- A *move generator*, which accepts a valid board and generates all boards that can be obtained by valid moves from that board, according to the rules of the game.
- A *static evaluation function*, which accepts a board and computes a value that indicates to which extent the board is good for the player who is to make a move.
- A *search engine* which selects a board and decides whether it should use the move generator to expand the tree, or the evaluation function to assign a value to leaf nodes. The programmer decides which search algorithm to use, for example *alpha-beta* or *IDA** search.
- *Search enhancements* that improve the quality of the search, such as a transposition table and the history heuristic.

The move generator is created by the Multigame compiler from a Multigame program. The evaluation function is also provided by programmer; this is a function written in C. The other parts are contained in the Multigame runtime system.

Automatic parallelism could be exploited in all three parts of the program, but only the search engine is suitable for coarse grained parallel architectures. In this document, however, we will further ignore the different aspects of parallelism, because all parallelism is implicit, thus the Multigame programmer need not worry about it. Design issues of the language and the implementation of the compiler and runtime system are discussed in [2].

The remainder of this document is structured as follows. Section 2 gives an informal introduction of the language. Section 3 describes the entire language in detail. Section 4 describes how a Multigame playing program reads and writes boards and how it communicates moves. The interface of the evaluation function is described in Section 5. Appendix A gives example programs for *tic-tac-toe*, *Connect-4*, the *15-puzzle*, *Othello*, *checkers*, and *chess*.

[1] The game of *life* can be expressed, although strictly speaking it is a zero-player game, since the sequence of moves from the starting position is fixed. It can be implemented as a one or two-player game, for which a player always has the "choice" of a single move.

2 The language principles

A Multigame program consists of a number of declarations, followed by a number of functions. The declarations are used to specify the number of players, the size of the board, etc. The functions contain statements that describe the legal moves. Each statement accepts a set of input boards, applies an operation to each of the boards in the set and puts the resulting boards into the output board set. The output board set of a statement is the input board set of the next statement.

A Multigame program requires a definition of the function "*main*". It accepts a (singleton) board set and produces the set of boards that can be reached by doing a move.

The language has a *Logo*-like paradigm [1]: there is a cursor that can be moved over the fields of a board. We call this cursor the *finger*. The finger points at one of the fields of the current board. Many statements implicitly use or modify the finger.

To describe the legal moves, Multigame uses several implicit variables:

- There is a *current board state*, which contains a matrix of *fields*. Fields are ordered in *rows* and *columns*. Each field can hold at most one *piece*. Whether white or black is to make a move (in two-person games) is part of the board state.
- There is a cursor which we call the *finger*, pointing at one of the fields.
- The *current direction* can be set to north, northeast, east, and so on. A **step** statement moves the finger one field in the current direction.
- Finally, there is a *hand*, which can temporarily hold a piece. A **pick up** statement removes the piece from the field currently pointed at by the finger. The piece is held in the hand while the finger can be moved across the board, until a **put down** statement is performed.

2.1 Some examples

The rules of a certain game describe the legal moves from a position. The following artificial examples illustrate how the language works.

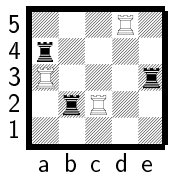
```
dimensions (5,5)

pieces
{
    rook 'R' 'r'
}

main = find rook,
       pick up,
       north,
       step,
       put down.
```

This example shows that the game is played on a 5×5 board, that there is one piece called "rook", and how to move a rook forward. The *'R'* and *'r'* are used for displaying the board on an ASCII

terminal.



Now look at an example input board for this game. Assume that white is the player to move. The first statement **find** *rook* searches for all white rooks. In this case, there are three white rooks, so it creates three copies of the board.[2] Each copy has its finger initialized to a different rook. From now on, we continue with three different, independent boards.

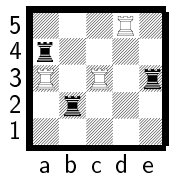
The second statement **pick up** continues with each of these boards and takes the current rook from the board. This piece is kept in the hand, and will be put down later using **put down**.

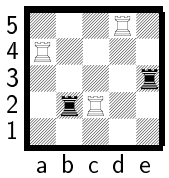
Before stepping, we need to set the current direction of each board. This is done using **north**; it initializes the current direction to "north". The actual step is done at the fourth statement by moving the finger one field in the current direction. The example illustrates the following cases.

- The board with the finger pointing at `c2` now has the finger pointing at `c3`.
- The finger pointing at `a3` now points at the black rook at `a4`. A finger can point at any[3] field, regardless whether it holds a black or white piece, or no piece at all. This does not (yet) cause the black rook to disappear from the board.
- The finger pointing at `d5` falls of the board. We do not allow pieces to fall off the board, thus the command fails and the board does not appear in the output set.

Now we proceed with two boards and execute **put down**. In the case of the board with the finger pointing at `a4`, the black rook is replaced by the white rook, which we still had in our hand. This is the normal way to implement capturing.

So we see that with the rules of this game, the above example yields two boards:





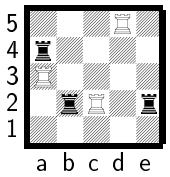
We assumed that white was to move, but now consider what happens if black is to move, using the same example.

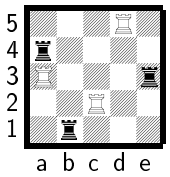
Black is supposed to be on the other side of the table (because of the absence of a **sides** declaration, see Section 3.4.3). From black's point of view, "north" is in the opposite direction, so black will move one of its rooks in the downward direction. Note that black is also allowed to make a capture move.

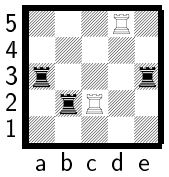
The result set for black is:

[2]The copying is conceptual; the Multigame compiler generates code that does backtracking.

[3]Except for unused fields, see Section 3.4.4.







2.2 The real rook move

We saw how to specify the rules that allow a rook to move one field forward, but in *chess*, the rook may move in all the four orthogonal directions and it is also allowed to move more than one field. However, it may not jump over other pieces (we do not consider castling in this example) and it may capture opponent's pieces only. Now we are going to modify our program so that the rook behaves the same as in *chess*.

First, we have to change the line which specifies the direction **north**. This can be done using the **either** construct:

```
main = find rook,
       pick up,
       either north or west or south or east,
       step,
       put down.
```

but because of its common use, there exists an abbreviation:

```
main = find rook,
       pick up,
       orthogonal,
       step,
       put down.
```

The effect of the **orthogonal** statement is that each input board is copied four times, and that for each board the copies have their current directions initialized to north, west, east, or south, respectively.

The second modification is harder, because the rook may move more than one field, but not jump over other pieces, although it may capture an opponent's piece. We therefore use the **repeat** statement. The **repeat** statement allows a block of statements to be repeated a certain amount of times. In its simplest form, this number is fixed, as in

```
repeat 3 times step.
```

which is equivalent to

```
step, step, step.
```

The power of the repeat statement comes from the fact that a piece of code can be repeated over a *range of times*. The statement

```
repeat 2 .. 4 times step.
```

could have been written otherwise using the **either** construction (The **[** and **]** brackets are used for grouping) :

```
either [ step, step ]
    or [ step, step, step ]
    or [ step, step, step, step ].
```

So we see that in this example the number of output boards is at most three times the number of input boards. *At most*, because the **step** command might fail. Suppose that the finger could be moved two fields, but when it does a third step, it would point outside the board. Then a fourth step is also impossible, so the iteration stops after applying two steps. For this particular input board, only one output board would be generated.

To express that the **step** command should be repeated one or more times, the keyword **infinity** is specified as upper bound. This does not mean that the step will be executed infinitely, because the finger will eventually step off the board and fail. The example

```
repeat 1 .. infinity times step
```

is still not what we want, because now we are able to jump over any piece and we can even capture our own pieces. To solve these problems, we reformulate the rook move: a rook must make a step, slide zero or more times over an empty field, and must finally point at an empty field (non-capture move) or an enemy piece (capture move). The following example formalizes this rule:

```
main = find rook,
       pick up,
       orthogonal,
       step,
       repeat 0 .. infinity times [ points at empty field, step ],
       either points at empty field or points at opponent's piece,
       put down.
```

The **points at empty field** statement fails if the finger points at a field that is not empty. Just like the **step** statement, it cuts off the iteration of the **repeat** statement if it fails.

Note that we can reformulate "points at an empty field or an enemy piece" as "does not point at an own piece". The **not** statement serves for this purpose, so we can make our code more compact and get this:

```
main = find rook,
       pick up,
       orthogonal,
       step,
       repeat 0 .. infinity times [ points at empty field, step ],
       not points at own piece
       put down.
```

To structure a Multigame program, the rules can be split in several functions. The example shows only how to move a rook, but there could have been other pieces whose moving rules are preferably defined in other functions. It does not matter in which order the different functions are defined.

```
main      = either rook_move or king_move.

rook_move = find rook,
            pick up,
            orthogonal,
            step,
            repeat 0 .. infinity times [ points at empty field, step ],
            not points at own piece
            put down.

king_move = ...
```

We saw that many statements (e.g., **either** ... **or** ... , **not** ...) have arguments that are statements (or statement sequences) itself. Grammatically, a sequence of statements, surrounded by square brackets, is a statement. Statements are always type compatible; all statements implicitly accept an input board set and implicitly produce an output board set. Thus every statement can be argument of a statement like **either** or **not**, and arbitrary complex rules can be built.

2.3 Winning, drawing, and losing a game

The rules of a game describe how a game is won, lost, or drawn (in two-player games). In general, there are two (not necessarily disjunct) ways to end a game:

- The player who is to make a move cannot make any legal move. The game ends with a win, draw or loss, depending on the rules.
- A certain pattern appears on the board, such as “four in a row”.

Before we know how to deal with a won, drawn or lost game, we need to detect such a situation. Therefore, we are going to look at some more commands that are often used for that purpose.

Let us begin with the **try** statement. Many games have rules like “if you can do this, you *must* do so”, possibly followed by “otherwise you must do something else”. The **try** statement is similar to if-statements in other languages. However, other languages have a separate if-condition followed by a then-statement. In Multigame, they are melt together in one block.

With this statement, we can specify that a game ends in a draw if a player cannot make a move:

```
main       = try legal_move else draw.

legal_move = ...
```

If the player cannot make a legal move, *main* returns a singleton output set, and the board has been marked that the game has ended with a draw. We do this, because the function *main* may not return an empty set; a runtime error occurs if the input position fails to produce at least one output board.

Another common sequence is used for the case that a game is won if a certain pattern is recognized:

```
main = legal_move,
       try [ test four_in_a_row, win ].
```

Here we introduce the **test** statement. In fact, it is the counterpart of the **not** statement, which we have seen before. For each board in the input, **test** fails if its argument fails, but continues with *the original board state* if its argument produces a non-empty set. Omitting the **test** would continue with the board state(s) as left by the argument.

It is important to understand the difference between **try**, **test** and **not**. Therefore, we summarize their behavior in the following table. B is the input board set. Note that for simplicity, we assume that B is a singleton. For the general case where B consists of multiple boards, the rules below are applied to each board in B separately. The following table shows the output sets for the **try**, **test**, and **not** statements for the cases that the argument S_1 results in an non-empty or empty set, respectively.

Statement	$S_1(B) \neq \emptyset$	$S_1(B) = \emptyset$
try S_1	$S_1(B)$	B
try S_1 **else** S_2	$S_1(B)$	$S_2(B)$
test S_1	B	$\emptyset$
not S_1	$\emptyset$	B

There are many ways to define *four_in_a_row*. One of them is as follows:

```
four_in_a_row = any direction,
                optionally [ step backward, points at own piece ],
                repeat 3 times [ step, points at own piece ].
```

We assume that the finger still points at a newly placed mark, and that we want to test whether it is in a line of four consecutive pieces of our own color. If it is, the newly placed piece is either on one the end of such a row, or next to the end. The **optionally** statement continues both with and without applying its argument. If the newly placed piece happens to be next to the end of a row, the row will be found because the finger will have been moved to either end of the row by the argument of the **optionally** statement. If the newly placed piece was already on the end of the row, the row will also be found because the **optionally** statement also succeed without applying its argument.

The **optionally** statement does not work for testing five-in-a-row. One way to test five-in-a-row is to keep on walking until an end of the row is reached, and test whether the row is long enough (this need not be done in all directions):

```
five_in_a_row = either north or northeast or east or southeast,
                while [ step backward, points at own piece ] do nothing,
                repeat 4 times [ step, points at own piece ].
```

For each board in the input set, the **while** statement alternately applies its two arguments (the *condition* and the *body*), until the condition fails. When the condition fails, the side effects causes by the (last) execution of the condition are undone, and execution resumes at the next statement. Thus, if the last step backward in the example succeeded but did not point at an own piece, the step backward is undone. If the body of a **while** statement fails, the entire **while** statement fails. In the example, the body cannot fail, since the **nothing** statement always succeeds.

To test very long sequences, it may be more efficient to start looking in two opposite directions and count the lengths explicitly using *properties*, which are explained in Section 2.4.

Games can also end after a particular board state is reached. The **match** statement tests all fields for a particular state. The following example shows how one can check in the *8-puzzle* whether the target

state is reached:

```
target_reached = try [
                     match ( empty field, piece_1, piece_2,
                                 piece_3, piece_4, piece_5,
                                 piece_6, piece_7, piece_8 ),
                     win
                 ]
```

The pieces are listed left to right, top to bottom.

2.4 Expressions and properties

Some games have rules that require extra board state to express game-specific properties like castling and en-passant permissions in *chess*. Therefore, Multigame supports user definable *properties* that can be used to determine whether a rook has already been moved, and so on. A property is some kind of named variable, associated with a board or a player. Currently, the only type that is supported is a subrange *integer*. For most purposes, this is sufficient.

Let us take a look at an example. In *Othello*, the game ends as soon as no player can place a new piece, i.e. if one player cannot do a move, the player is forced to pass, but if the opponent cannot do a move either, the game has ended. We therefore need a boolean variable that remembers whether a player passed in its last move. We define a board's property *passed* as a boolean integer as follows:

```
properties
{
    passed : board : [ 0 .. 1 ].
}
```

Note that the **properties** declaration should immediately follow the **pieces** declaration. A typical piece of program code that uses this property is this:

```
main = try [ legal_move, assign (passed = 0) ]
       else try [ assert (!passed), assign (passed = 1) ]
       else end_game.
```

Two new statements are introduced here, **assign** and **assert**. Both statements require a C-like expression. **assign** just evaluates the expression, but **assert** also checks for a non-zero result. If the expression evaluates to zero, the statement fails.

The code above shows three possibilities. If the current player can make a legal move (i.e. *legal_move* produces a non-empty set), the property *passed* is set to zero. If *legal_move* fails, it tests *!passed* to see whether the previous move (by the other player) resulted in a pass. If this is not the case, it assigns one to *passed* to indicate that the player has passed. However, if the assertion fails, the whole try-block fails and the function *end_game* is called.

We have not yet defined the function *end_game* yet. It should count the pieces to determine which player owns most pieces. To implement this, we need a variable that stores the number of own pieces minus the number of opponent's pieces. We declare this the following way:

```
properties
{
    passed : board     : [   0 ..  1 ].
    score  : temporary : [ -64 .. 64 ].
}
```

The property *passed* is persistent between moves, because it is part of the board state. This kind of properties is used to test equivalence between boards. If at runtime the program compares two boards with each other, and both boards have the same pieces at the same fields, the same player to make the next move, but differ in property value(s) only, the boards are not considered equal.

In the previous example, we do not want *score* to be persistent between moves. We therefore declare the property to be **temporary**,[4] to ensure that *score* is only alive within one move and not between moves.

As an illustration, we also show the code that implements the *end_game* function:

```
end_game = assign (score = count(own piece) - count(opponent's piece)),
           try [ assert (score > 0), win ]
           else try [ assert (score < 0), loss ]
           else draw.
```

A property can also be associated with each **player**. It is much like the board's property (like *passed*) but private copies for both the white and the black player are kept. This is useful if one wants to record separate rights for both players, such as castling rights in *chess*.

```
properties
{
    may_castle_left, may_castle_right : player : [ 0 .. 1 ].
}
```

This eases programming, because the programmer need not write separate code for the black and the white player (although it can be done). For example, modifying *may_castle_left* only influences the value for the current player, but not for the opponent, except if the name of the property is prefixed with **opponent's**, **white's**, or **black's**. Player's properties are persistent between moves. Player's properties are not allowed in one-player games; board's properties can be used instead.

2.5 Other useful statements

There are some statements that have not been described yet, but are worth to be mentioned here. For a full description, see the section about the formal description of the language.

nothing does nothing. Occasionally the Multigame syntax requires it.

irreversible states that this move causes a conversion, i.e. that in the remainder of this game, it is not possible that this board will appear again. It is not obligatory, but could improve the quality of the generated code.

[4] This is more a quick hack in the current compiler than an elegant way of implementing variables. **temporary** variables should not be declared in the **properties** section; in fact they are not properties at all. They should be locally declared, but to be useful, this also requires the implementation of call-by-reference argument passing to other functions.

move to ... and **any position** are alternate ways to initialize the finger.

turn ... is a simple way to change the current direction.

replace by ... is a useful statement to introduce new pieces and an alternative way to implement capturing.

white to move and **black to move** provide a way to implement rules that are different for the white and the black player. It is suggested to use them within a **try** or an **either** statement.

optionally ... is equivalent to **either nothing or** ... .

while ... **do** ... and **do** ... **while** ... loops are useful for iterating statements.

all positions ... applies its argument to all positions. **all directions** ... applies its argument in all directions.

restore position after ... restores the finger after applying its argument so that it points at the field before the argument was applied. **restore direction after** ... restores the direction after applying its argument to the direction before the argument was applied.

3 Reference manual

3.1 Introduction

This section describes the Multigame language formally.

A Multigame program describes the rules of a board game. From a Multigame program, a Multigame compiler generates a move generator that is used by a game-playing runtime system. The move generator's task is to generate the list of legal moves from a give position.

The Multigame language is restricted to handle the following class of games:

- The game must be a *board game*.
- The game is either for one or for two players.
- The board consists of a two-dimensional, rectangular grid of fields. Not all fields need to be used, and conceptual transformations from non-rectangular boards are often possible.
- Perfect information is required (i.e. no pieces with hidden identities are allowed).
- Games that depend on chance or probability are excluded.

The language is based on a combination of the Logo and Prolog programming paradigms. The language has implicit knowledge about the concepts "board", "field", "piece", "player", and "move". To describe a move, Multigame uses several implicit variables:

- There is a *current board state*, which contains a matrix of *fields*. Each field holds at most one *piece*. Whether white or black is to make a move (in two-person games) is considered part of the board state.
- There is a cursor which we call the *finger*. The finger points at one of the fields of the current board. Many statements implicitly use or modify the finger.
- The *current direction* can be set to north, northeast, east, and so on. A **step** statement moves the finger one field in the current direction.
- There is a *hand*, which can temporarily hold a piece. A **pick up** statement removes the piece from the field currently pointed at by the finger. The piece is held in the hand while the finger can be moved across the board, until a **put down** statement is performed.

Each statement accepts a set of boards as input and applies a test or modification to each position in the input set. For each position for which the statement succeeds, the resulting boards are placed in the output set (some statements can succeed in multiple ways and yield multiple boards). The output board set of a statement is the input board set for the next statement.

The programmer can define functions to structure a Multigame program. Each function has a set of boards as an implicit formal argument, and implicitly returns the set of boards that results from applying the statements to the actual argument. Functions are allowed to be recursive. The programmer

must provide a function *main*. The Multigame runtime system calls this function with a singleton input position and expects it to return the set of boards that can be reached by doing a legal move.

3.2 Lexical conventions

White space consists of spaces, tabs, and newlines. White space separates lexical tokens.

A <*character-constant*> is an apostrophe, followed by a printable character and another apostrophe.

An <*integer*> is a sequence of one or more digits.

An <*identifier*> starts with a letter or underscore and is followed by zero or more letters, underscores, or digits. The following identifiers are reserved keywords (note that the keywords with an apostrophe are not identifiers):

after	**all**	**any**	**anyone's**	**assert**	**assign**
at	**avoid**	**backward**	**black**	**black's**	**board**
by	**column**	**count**	**diagonal**	**dimensions**	**direction**
directions	**do**	**down**	**draw**	**east**	**either**
else	**empty**	**exchange**	**field**	**find**	**forward**
infinity	**irreversible**	**layout**	**loop**	**loss**	**match**
north	**northeast**	**northwest**	**move**	**not**	**nothing**
opponent's	**optionally**	**or**	**orthogonal**	**own**	**pick**
piece	**piecenumber**	**pieces**	**player**	**players**	**points**
position	**positions**	**properties**	**put**	**repeat**	**replace**
restore	**row**	**sides**	**southeast**	**south**	**southwest**
step	**symmetry**	**temporary**	**test**	**times**	**to**
try	**turn**	**west**	**while**	**white**	**white's**
win					

It is recommended to avoid using the following identifiers, since they may be used in the future: **case**, **declare**, and **switch**.

3.3 The Multigame syntax

<*program*> ← <*declarations*> <*rules*>

A Multigame program consists of a number of declarations, followed by a number of rules.

3.4 Declarations

<declarations> ← *<players>* ?
<board-size>
<nr-sides> ?
<board-layout> ?
<symmetry-declaration> ?
<pieces-declaration>
<properties> ?

A Multigame program starts with a number of declarations. Only the dimensions of the board and the names of the pieces are obligatory. The order of the declarations is fixed.

3.4.1 Number of players

<players> ← **players '='** *<integer>*

This declaration specifies the number of players. *<integer>* must be 1 or 2. If the declaration is omitted, the game is for two players.

3.4.2 Board size

<board-size> ← **dimensions '('** *<expression*$_1$*>* **','** *<expression*$_2$*>* **')'**

This declaration defines the size of the board. *<expression*$_1$*>* defines the width or the board, and *<expression*$_2$*>* the height. Both must be greater or equal to 1. The declaration is obligatory.

3.4.3 Board orientation

<nr-sides> ← **sides '='** *<integer>*

This declaration specifies in two-player games whether the players are on opposite sides of the board. **sides = 2** states that the white player sits on the lower side of the board and the black player on the upper side. As a convention, both players see the origin of the board, with coordinates (1,1), on their lower left-hand side of the board. This implies that the origin for the black player is the opposite corner field of the origin for the white player. Moreover, directions are reversed: what **north** means for one player means **south** for the other. **sides = 1** states that both players sit on the lower side of the board, and see the board from the same perspective. If the declaration is omitted, both players are on the opposite sides. The declaration is not allowed in one-player games.

3.4.4 Board layout

<board-layout> ← **layout '{'** [**'*'** | **'.'** | **'–'**] + **'}'**

In some games, such as *checkers*, some fields are never used. It is possible to describe which fields

will never be used, by drawing an approximation of the board layout. A **'*'** means that the field may hold a piece, but fields marked with **'.'** or with **'–'** will always be empty. A field marked with **'–'** can be stepped through, while a field marked with **'.'** cannot, as explained in Section 3.5.7.

```
layout
{
    .*.*.*.*
    *.*.*.*.
    .*.*.*.*
    *.*.*.*.
    .*.*.*.*
    *.*.*.*.
    .*.*.*.*
    *.*.*.*.
}
```

shows the fields used with 8×8 *checkers*.

If <*board-layout*> is specified, it should contain exactly *width* $\times$ *height* dots, asterixes and minus signs. If it is omitted, all fields default to **'*'**, i.e. each field can hold a piece.

3.4.5 The symmetry declaration

<*symmetry-declaration*> ← **symmetry** <*symmetry-lines*>

<*symmetry-lines*> ← **all directions** | **orthogonal** | **diagonal** |
[<*direction*> **'–'** <*direction*>]

<*direction*> ← **north** | **northeast** | **east** | **southeast** | **south** | **southwest** | **west** | **northwest**

The symmetry declaration is used to express that a board is always symmetrical in some lines, i.e. that boards that are each other's mirror image always have the same evaluation value. For example, with *tic-tac-toe* there are four symmetry lines: horizontal, vertical and diagonal. This can be expressed as follows:

```
symmetry all directions
```

If there is only one symmetry line, for example the vertical one, it can be expressed as:

```
symmetry north–south
```

It is not obligatory to express existing symmetry lines, but by doing so, more efficient code can be generated.[5]

[5] The current compiler does not use this information.

3.4.6 The pieces declaration

<*pieces-declaration*> ← **pieces** **'{'** <*piece-declaration*> + **'}'**

<*piece-declaration*> ← <*identifier*> <*character-constant*$_1$> <*character-constant*$_2$> ?

Declares the names and representations of the pieces. A piece name <*identifier*> may not be a *field number*, i.e. an lower or upper case letter followed by one or more digits. The character constants are used as representation for a white and a black queen, respectively (see Section 4). <*character-constant*$_2$> must only be specified in two-player games and may not be specified in one-player games. A character constant may be used once.[6]

3.4.7 Properties

<*properties*> ← **properties** **'{'** <*property-declaration*> * **'}'**

<*property-declaration*> ← <*property-names*> **':'** <*property-owner*> **':'** <*property-type*>

<*property-names*> ← <*identifier*> [**','** <*identifier*>] *

<*property-owner*> ← **board** | **player** | **temporary**

<*property-type*> ← **'['** <*constant-expression*$_1$> **'..'** <*constant-expression*$_2$> **']'**

A *property* is a kind of named variable, associated with some *owner*, and of a particular *type*. The name spaces for properties and pieces are separate.

A property can be associated with a *board*, a *player*, or can be *temporary*. A property that is associated with a board extends the board's state; two boards with the same pieces on the same fields but with different board property values are considered different. A property that is associated with a player is like a property that is associated with a board, but separate values for the white and the black players are maintained. Player properties are not allowed in one-person games. A temporary property does not extend the board state; it is a global variable with a lifetime that spans the time to compute all moves from a position. Board and player properties persist between moves.[7] In a future version of the language a property may also be associated with a *piece*.

A property is of a certain type; currently only subrange integers are supported. This may be extended with position, direction, piece, and board types in the future.

All properties are declared in a single <*properties*> declaration. If no properties need be declared, the <*properties*> declaration can be omitted. Multiple properties with the same owner and of the same type may be declared in a single <*property-declaration*>.

Properties can be used in **assign** and **assert** statements (see Section 3.5.12).

[6] This currently imposes a limit on the number of different pieces.

[7] The name "temporary property" is a misnomer and its adoption in the language is inelegant, but is introduced to keep the language and a compiler implementation simple.

3.5 Rules

The last section of a Multigame program formally describes the legal moves.

<rules> ← *<function>* +

Each Multigame program contains one or more functions, which formally describe the movement and capturing rules of a game.

3.5.1 Functions

<function> ← *<identifier>* **'='** *<statement-sequence>* **'.'**

A function's name is specified by *<identifier>*; its body by *<statement-sequence>*. A function implicitly takes an input set of boards as formal argument, and implicitly returns an output set of boards. Explicit parameter passing is not possible; this might change in the future. The name spaces for function names, piece names, and property names are separate. It is not allowed to define a function twice.

Definition of the function "*main*" is obligatory. The Multigame runtime system calls *main* with a singleton board set: the position from which a move is to be made.

3.5.2 Blocks

<statement> ← **'['** *<statement-sequence>* **']'**

<statement-sequence> ← [*<statement>* **','** ?] +

A *block* is a sequence of statements, surrounded by square brackets. The comma at the end of each statement is optional. The board output set of a statement is the input board set of the following statement.

3.5.3 Piece descriptions

The statements **find** ... , **points at** ... and **replace by** ... as well as the expression **count(** ... **)** use the following grammar rules as argument:

<piece-description> ← *<owner>* ? [**piece** | *<identifier>*]

<piece-description> ← **empty field**

<owner> ← **own** | **opponent's** | **anyone's** | **white's** | **black's**

The *<owner>* part is optional; the default is **own**. If **piece** is given, any piece of the appropriate color matches. Otherwise *<identifier>* must be a piece name declared in the **pieces** declaration. **anyone's piece** matches every piece but it does not match an empty field.

3.5.4 Initializing the finger

There are several commands that initialize the finger. The behavior of a command which uses an uninitialized finger is undefined. The Multigame compiler may produce an executable that tests the validity of the finger, so that it can print a runtime error, but it need not do so.

<*statement*> ← **move to** <*position-expression*>

<*position-expression*> ← **'('** <*expression*$_1$> **','** <*expression*$_2$> **')'**

<*position-expression*> ← **position**

Initializes the finger to point at the field denoted by <*position-expression*> for each board in the input set. If the finger points outside the board or at a field that is marked as "never used" in the <*board-layout*> declaration, the board is discarded from the output set.

A <*position-expression*> can be constructed from a column number (<*expression*$_1$>) and a row number (<*expression*$_2$>). See Section 3.4.3 for a discussion about the coordinates of the board.

position denotes the current position, and is not useful in a **move to** ... statement, but is there for future extensions in the Multigame language.

<*statement*> ← **any position**

Creates for each board in the input set as many board as needed, such that each copy has its finger initialized to a different field. There are no boards created with a finger pointing at a field marked as "not used" in the <*board-layout*> declaration.

<*statement*> ← **find** <*piece-description*>

Equivalent to **[any position , points at** <*piece-description*> **]** (see Section 3.5.6).

<*statement*> ← **all positions** <*statement*$_1$>

Applies <*statement*$_1$> to all used fields. The order in which the finger is iterated is unspecified. Only succeeds if all iterations succeed. After all iterations are performed, the finger is restored to the original position. Often used to set all fields of the board.

3.5.5 Setting the direction

<*statement*> ← **north** | **northeast** | **east** | **southeast** | **south** | **southwest** | **west** | **northwest**

Sets the current direction to "north", "north-east", "east", "south-east", "south", "south-west", "west", or "north-west", respectively.

<*statement*> ← **any direction**

any direction copies each board 8 times, and initializes their current directions to “north”, “north-east”, “east”, “south-east”, “south”, “south-west”, “west”, or “north-west”, respectively.

<*statement*> ← **diagonal**

diagonal copies each board 4 times, and initializes their current directions to “north-east”, “south-east”, “south-west”, or “north-west”, respectively.

<*statement*> ← **orthogonal**

orthogonal copies each board 4 times, and initializes their current directions to “north”, “east”, “south”, or “west”, respectively.

<*statement*> ← **turn** <*constant-expression*>

Turns the current direction of all boards <*constant-expression*> degrees. The angle must be a multiple of 45. A positive value indicates a counter-clockwise turn, a negative value a clockwise turn.

<*statement*> ← **all directions** <$statement_1$>

Applies <$statement_1$> in all directions. Restores the finger to the original value each time <$statement_1$> is applied. The order in which the direction changes is unspecified. Only succeeds if <$statement_1$> succeeds all times. The direction is undefined after the statement succeeds.

3.5.6 Testing pieces

<*statement*> ← **points at** <*piece-description*>

Succeeds if the finger is pointing at a piece matching <*piece-description*>; fails otherwise.

3.5.7 Stepping

<*statement*> ← **step** [**backward** | **forward**] ?

Moves the finger one field in the current direction (or in the opposite direction if **backward** is appended). Fails if, after stepping, the finger points outside the board or if the finger points at a field marked with **’.’** in the <*board-layout*> declaration. If, after stepping, the finger points to a field marked with **’–’**, the step is *repeated* until the finger points outside the board or at a field marked with **’.’** (the statement fails in both cases), or at a field marked with **’*’** (in which case it succeeds).

3.5.8 Moving pieces

<*statement*> ← **pick up**

Picks up the piece currently pointed at by the finger and keeps it in the hand. After picking it up, the field is made empty. If the field was already empty, the hand holds an “empty field”.

<*statement*> ← **put down**

Puts down the piece held in the hand (obtained by the most recent **pick up** or **exchange**) at the field currently pointed at by the finger. If there was already a piece at that field, it is captured (i.e., removed from the board). If the hand holds an "empty field", the field is cleared. Multiple successive **put down** statements are allowed but do not stack, thus in the sequence

```
find king, pick up,
find queen, pick up,
put down,
put down,
```

both **put down** statements put down a queen. The behavior is undefined if no preceding **pick up** has been performed.

<*statement*> ← **exchange**

Exchanges the piece pointed at by the finger and the piece held in the hand. Either may hold an "empty field". The behavior is undefined if no preceding **pick up** has been performed.

<*statement*> ← **replace by** <*piece-description*>

Puts a piece at the field pointed by the finger. If the piece name in <*piece-description*> is an identifier, the field is replaced by a piece with the given name, with the color given in <*piece-description*> (default is **own**). **anyone's** is not allowed in <*piece-description*>. If **piece** is specified instead of a piece name, the color of the piece currently pointed at is set to the color given in <*piece-description*> (again, **anyone's** is not allowed), or does nothing if it was an empty field. **replace by empty field** clears the field.

3.5.9 Function invocation

<*statement*> ← <*identifier*>

Calls function <*identifier*>. It is not necessary for the definition of the callee to textually precede the point at which the function is called. Functions may be recursive.

3.5.10 Conditional execution

<*statement*> ← **test** <$statement_1$>

Each board in the input set is applied to <$statement_1$>. If the result is a non-empty set, the *original* board is placed in the output set, thus all side effects in <$statement_1$> are undone. Fails if <$statement_1$> produces an empty set.

<*statement*> ← **not** <$statement_1$>

Each board in the input set is applied to <*statement*$_1$>. If the result is an empty set, the *original* board is placed in the output set, thus all side effects in <*statement*$_1$> are undone. Fails if <*statement*$_1$> produces a non-empty set.

<*statement*> ← **try** <*statement*$_1$>

Each board in the input set is applied to <*statement*$_1$>. If <*statement*$_1$> is non-empty, this result is placed in the output set; otherwise the original input board is placed in the output set. Always succeeds.

<*statement*> ← **try** <*statement*$_1$> **else** <*statement*$_2$>

Each board in the input set is applied to <*statement*$_1$>. If <*statement*$_1$> is non-empty, this result is placed in the output set. Otherwise, it executes <*statement*$_2$> instead and places this result in the output set. There is an ambiguity in the **try** / **try-else** grammar rules: it is resolved by having the else-part belong to the closest preceding try-part.

3.5.11 Restoring the finger or direction

<*statement*> ← **restore direction after** <*statement*$_1$>

Executes <*statement*$_1$>. The direction is then restored to the value before <*statement*$_1$> is applied.

<*statement*> ← **restore position after** <*statement*$_1$>

Executes <*statement*$_1$>. The finger is then restored to the value before <*statement*$_1$> is applied.

3.5.12 Expressions

Expressions are almost the same as in C. The sizeof, comma, dereference, address, and index operator are not supported.

<*expression*> ← <*owner*> ? <*identifier*>

<*identifier*> must be a property, declared in the **properties** section. If the property is a player's property, <*owner*> (see Section 3.5.3) may be prefixed to select the player (default is **own**). It is not allowed to prefix a property with **anyone's**. A property can be used as rvalue or as lvalue.

<*expression*> ← <*integer*>

<*expression*> ← **count '('** <*piece-description*> **')'**

Counts the number of pieces that match <*piece-description*>. Note that **count (empty field)** does not count the fields that are marked as "never used" in the <*board-layout*> declaration. May not be used as lvalue.

<*expression*> ← **row** | **column**

Gives the current row resp. column number. May not be used as lvalue.

<*expression*> ← [**row** | **column**] **'('** <*position-expression*> **')'**

Extracts the row resp. column number from <*position-expression*>. May not be used as lvalue.

<*expression*> ← **'('** <*expression*$_1$> **')'**

Used for grouping.

<*expression*> ← <*prefix-operator*> <*expression*$_1$>

<*expression*> ← <*expression*$_1$> <*infix-operator*> <*expression*$_2$>

<*expression*> ← <*expression*$_1$> <*postfix-operator*>

<*expression*> ← <*expression*$_1$> **'?'** <*expression*$_2$> **':'** <*expression*$_3$>

<*prefix-operator*> ← **'!'** | **'~ '** | **'+'** | **'–'** | **'++'** | **'– –'**

<*infix-operator*> ← **'*'** | **'/'** | **'%'** | **'+'** | **'–'** | **'<<'** | **'>>'** | **'<'** | **'<='** | **'>'** | **'>='** | **'=='** | **'!='** | **'&'** | **'^ '** | **'|'** | **'&&'** | **'||'** | **'='** | **'*='** | **'/='** | **'%='** | **'+='** | **'–='** | **'<<='** | **'>>='** | **'&='** | **'|'** | **'^ ='**

<*postfix-operator*> ← **'++'** | **'– –'**

Ambiguities are resolved in the same way as in C.

<*statement*> ← **assign** <*expression*>

Evaluates <*expression*> for each board in the input set. Always succeeds.

<*statement*> ← **assert** <*expression*>

Evaluates <*expression*> for each board in the input set. Fails if the expression evaluates to zero; succeeds otherwise. If <*expression*> is a postfix increment or decrement, the test for zero is performed before the increment or decrement takes place.

3.5.13 Multiple choices and loops

<*statement*> ← **either** <*statement*$_1$> [**or** <*statement*$_n$>] *

Specifies that exactly one of the alternatives is executed. The whole input set is copies *n* times, and these sets are independently of each other applied to the different alternatives. The statement succeeds for any succeeding <*statement*$_i$>.

<*statement*> ← **repeat** <*repeat-range*> **times** <*statement*$_1$>

<*repeat-range*> ← <*integer*$_1$>

<*repeat-range*> ← <*integer*$_1$> **'..'** [<*integer*$_2$> | **infinity**]

Specifies that a rule should be applied multiple times. In it simplest form, <*statement*$_1$> is repeated a fixed number, <*integer*$_1$>, of times. If a range of repetitions is given, the output set is defined as the union of all sets produced by applying <*statement*$_1$> *i* times, for all *i* such that <*integer*$_1$> $\leq i \leq$ <*integer*$_2$>, or for all *i* such that <*integer*$_1$> $\leq i$ if **infinity** is given as upper bound. In the latter case, <*statement*$_1$> should fail after some iterations or the move generator will run out of stack space. <*integer*$_1$> may be zero. <*integer*$_2$> may not be smaller than <*integer*$_1$>.

<*statement*> ← **optionally** <*statement*$_1$>

Equivalent to **either nothing or** <*statement*$_1$>, or equivalently, **repeat** 0 .. 1 **times** <*statement*$_1$>. Intended to state that the player is allowed, but not obliged to apply a rule.

<*statement*> ← **while** <*statement*$_1$> **do** <*statement*$_2$>

Each board in the input set is alternately applied to <*statement*$_1$> and <*statement*$_2$> If <*statement*$_1$> is the first to fail, the side effects caused by the last execution of <*statement*$_1$> are undone, and execution is resumed at the next statement. If <*statement*$_2$> is the first to fail, the whole **while** statement fails, and the result does not appear in the output set. If <*statement*$_1$> or <*statement*$_2$> introduces new boards (by duplication), execution continues for each board separately.

<*statement*> ← **do** <*statement*$_1$> **while** <*statement*$_2$>

For each board in the input set, tries to execute <*statement*$_1$> and <*statement*$_2$> alternately. If <*statement*$_1$> fails first, the whole **while** statement fails, and the board disappears from the output set. If <*statement*$_2$> fails first, the side effects caused by the (last) execution of <*statement*$_2$> are undone, and execution is resumed at the next statement. If <*statement*$_1$> or <*statement*$_2$> introduces new boards (by duplication), execution continues for each board separately.

3.5.14 The final move

<*statement*> ← **win** | **draw** | **loss**

States that this board is a leaf in the search tree. Note that each leaf in the tree must be marked by either **win**, **draw** or **loss**. It is a runtime error if "*main*" returns an empty set; this would imply that the board given as input to "*main*" was a leaf node (no moves possible from this board), without having been marked as one.

3.5.15 Miscellaneous

<*match*> ← **match '(' <*piece-description-list*> ')'**

<*piece-description-list*> ← <*piece-description*> [**','** <*piece-description*>] *

Tests whether the board matches <*piece-description-list*>. Each <*piece-description*> should unambiguously specify one piece or **empty field**. If the board does not match, it is removed from the output set. The pieces should be listed from left to right, top to bottom, from the white's player perspective. Only fields that are actually used should be specified.

<statement> ← **[white | black] to move**

Only succeeds if the indicated player is making a move. It can be used to implement rules that are different for black and white.

<statement> ← **avoid loop**

Operates on the current field. Checks whether during this move an **avoid loop** statement has been executed before with the finger pointing at the current field. Fails if this is the case; succeeds otherwise. Often used in conjunction with repeating statements and recursive functions.

<statement> ← **irreversible**

States that the current move is a conversion, i.e. that in the remainder of this game, it is not possible that this board will appear again. It is not obligatory, but could improve the quality of the generated code. Always succeeds.

<statement> ← **nothing**

This statement does nothing. Occasionally the syntax requires it.

4 Input / Output

A game-playing Multigame program reads the opening board position from standard input or file, and can write the board obtained after doing the "best" move to standard output or file. The format is described in Section 4.1.

A game-playing Multigame program can also read and write the differences between the position from which a move is made and the position to which a move is made in an almost game independent way. The format is described in Section 4.2.

4.1 Board I/O

In this section we describe the format of the input and output boards. Both input and output have the same format, so that the output of one program can be piped into the input of another program. The format is human-readable; it contains no control characters.

The format is divided into three sections:

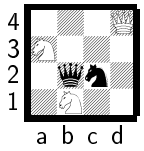
- Positions of the pieces,
- User defined properties,
- Player to make the next move.

In the **pieces-declaration** of a game (see Section 3.4.6), one or two character constants must be defined which state the representation of a white and a black piece, depending on the number of players. For example:

```
pieces
{
    knight 'N' 'n'
    queen  'Q' 'q'
}
```

declares a knight and a queen. A white knight is represented as *'N'* and a black knight as *'n'*. The *'.'* is reserved for denoting an empty field.

Now suppose we have some rules defined for this game and we want to compute black's best move from the following board:



Then we have to enter the following as input:[8]

[8]The current compiler generates code that is very strict about the input format. Even a spurious space causes the input to be rejected.

```
...Q
N...
.qn.
.N..
black to move
```

If a game defines board or player properties, they have to appear in the input and output as well. Suppose the game has defined the following properties:

```
properties
{
    property1, property2 : board     : [ 0 .. 2 ].
    player_property      : player    : [ 1 .. 3 ].
    temporary_property   : temporary : [ 2 .. 4 ].
}
```

Now an example of correct input would be

```
...Q
N...
.qn.
.N..
property1 = 0
property2 = 1
player_property = 2,3
black to move
```

Temporary properties do not appear in the input or output, since they their lifetime is limited to the generation of a move. Player properties have two values. The first denotes the value for the white player, the second for the black player. All properties appear in the order in which they are declared (see Section 3.4.7).

In two-player games the phrase “white to move” or “black to move” indicates the player to move; the phrase is omitted in one-player games.

4.2 Move I/O

In this section we give the format to describe a move. Both input and output have the same format. The format is human-readable.

Move descriptions are ambiguous: often multiple descriptions describe the same move. If the game-playing program prints a move description, it will print the shortest description that describes the move. If the player must enter a move, any correct move description will be accepted. The playing program will check entered moves against the rules.

The entire move description appears on a single line. Lexical tokens may be separated by spaces and tabs.

A <*field-number*> is a lower or upper case letter followed by an integer that indicates a position on the field. The letter specifies a column[9] and starts with ’A’ or ’a’. The number specifies the row and

[9] This currently restricts the number of columns.

starts with 1. A1 specifies the lower left-hand side corner field for white, as in chess.

An <*identifier*> starts with a letter or underscore and is followed by zero or more letters, underscores, or digits.

A <*piece-name*> is an <*identifier*> that is not a keyword and not a <*field-number*>.

The keywords are:

black	**draw**	**empty**	**field**	**loss**	**move**	**pass**	**white**	**win**

<*move*> ← **move** [<*sub-move*> **';'** +] +

A move starts with the keyword **move**, followed by a number of sub-moves, each separated by at least one semi-colon. Each sub-move describes part of the change in the board state. The fact that the player to move changes is implicit in two-person games and is not specified.

<*sub-move*> ← <*piece-move*>

<*sub-move*> ← <*property-change*>

<*sub-move*> ← <*terminal*>

<*sub-move*> ← **pass**

A sub-move describes a piece move, a property change, states that the move ends the game, or that nothing is changed.

<*piece-move*> ← <*source*> **'–>'** <*destinations*>

<*source*> ← <*field-number*> | <*piece-description*>

<*destinations*> ← <*field-number*> [**','** <*field-number*>] *

<*piece-description*> ← **empty field**

<*piece-description*> ← [**white** | **black**] ? <*piece-name*>

A piece move moves a piece to one or more fields. If a field number is specified as source, the piece currently on that field, if any, is removed and replaced by an empty field. The piece is copied to all destination fields. If the original field was empty, all destinations will become empty. If an explicit piece name is specified, the piece is copied to all destinations. If the color of the piece name is omitted, the color is the color of the player making a move.

<*property-change*> ← <*color-prefix*> ? <*identifier*> **'='** **'–'** ? <*integer*>

<*color-prefix*> ← [**white** | **black**] **'.'**

Assigns the value (–) <*integer*> to property <*identifier*>. <*identifier*> must be a board or player property. If <*identifier*> is a board property, specifying <*color-prefix*> is not allowed. If <*identifier*> is a player property and <*color-prefix*> is not specified, the property of the player making the move

is changed. If <*identifier*> is a player property and <*color-prefix*> is specified, the property of the indicated player is changed. The value must be within range, as specified in the property declaration of the game.

<*terminal*> ← **win** | **draw** | **loss**

States that the move ends the game, and leads to a win, draw (two-person games only), or loss for the player who is currently making a move.

Examples of move descriptions are given below:

`white mark -> b2;`	*tic-tac-toe*	draw cross in the middle
`a3 -> c5; empty field -> b4;`	*checkers*	capture move
`e1 -> c1; a1 -> d1;`	*chess*	queen-side castle white
`empty field -> e7; queen -> e8;`	*chess*	pawn promotion

5 Evaluation function interface

The programmer can provide an evaluation function to improve the quality of the game-playing program. The evaluation function is called through the C calling interface. The evaluation function accepts a position and analyzes it. In two-player games, the evaluation function should return a signed integer: a high value indicates that the position is advantageous for the white player and a low (negative) value indicates that the position is advantageous for the black player. In one-player games, the evaluation function should return a lower bound on the distance to a target position; the returned value should be as close as possible to the real distance, but must never overestimate the real distance.

The interface between the evaluation function and the Multigame runtime system is shown below:

```
typedef ...       field_type;

struct board {
    field_type    fields [NR_FIELDS];
    unsigned char white_to_move;       /* Two-player games only */
    ...
};

struct node {
    struct board  board;
    ...
};

int evaluate(const struct node *);
```

An evaluation function must be written in a separate file that starts by including **"node/node.h"** to include the declarations above. The Multigame compiler compiles and links the evaluation function when it is given the option **-eval=file.c**.

field_type is a scalar type that is large enough to represent all possible pieces. An empty field is represented by the value 0. Other pieces are numbered in the same order as they appear in the <*pieces-declaration*> in the Multigame program, starting from 1. In two-player programs, white pieces are positive and black pieces are negative. The black pieces are numbered downward, starting from minus 1.

struct board includes an array *fields* that represents the pieces on the board. The fields are numbered left to right, bottom to top, from the white player's perspective. Unused fields, as specified by the <*board-layout*> declaration, are not represented. *white_to_move* (only available in two-player games) is non-zero when white is to make a move from this position. The values of board and player properties are included in *struct board*. The evaluation function can access a board property (read-only) by reading its corresponding fields in the board structure. The compiler renames a board property by prepending *gcp_* to its name. The compiler renames a white player's property by prepending *gcpw_* to its name. The compiler renames a black player's property by prepending *gcpb_* to its name. All properties are scalar types which are sufficiently large to accommodate the entire range of legal values, as specified in their range declarations.

board is the only publicly accessible field from *struct node*.

A Example programs

A.1 Tic-tac-toe

```
dimensions (3,3)

symmetry all directions

pieces
{
    mark        'X' 'O'
}


main =          irreversible,           # each move is a conversion
                try new_mark else draw.

new_mark =      find empty field,
                replace by mark,
                try [ test three_in_a_row, win ].

three_in_a_row = find own piece,        # start from any position
                 any direction,
                 repeat 2 times [ step, points at own piece ].
```

A.2 Connect-4

```
dimensions (7,6)
sides = 1
symmetry north-south

pieces
{
    mark        'X' 'O'
}


main =          try new_mark else draw.

new_mark =      irreversible,
                south,
                find empty field,
                not [ step, points at empty field ],
                replace by mark,
                try winning_pos.

winning_pos =   any direction,
                optionally [ step, points at own piece ],
                repeat 3 times [ step backward, points at own piece ],
                win.
```

A.3 The 15-puzzle

```
players = 1
dimensions (4,4)
symmetry northwest-southeast

pieces
{
    piece_1     '1'
    piece_2     '2'
    piece_3     '3'
    piece_4     '4'
    piece_5     '5'
    piece_6     '6'
    piece_7     '7'
    piece_8     '8'
    piece_9     '9'
    piece_10    'A'
    piece_11    'B'
    piece_12    'C'
    piece_13    'D'
    piece_14    'E'
    piece_15    'F'
}


main =          move_piece,
                try goal_reached.

move_piece =    find empty field,
                orthogonal,
                step,
                pick up,
                step backward,
                put down.

goal_reached =  match (empty field, piece_1 , piece_2 , piece_3 ,
                       piece_4,     piece_5 , piece_6 , piece_7 ,
                       piece_8,     piece_9 , piece_10, piece_11,
                       piece_12,    piece_13, piece_14, piece_15),
                win.
```

A.4 Othello

```
dimensions (8,8)
symmetry all directions

pieces
{
    mark        'X' 'O'
}

properties
{
    passed              : board     : [ 0 .. 1 ]
    flipped_something   : temporary : [ 0 .. 1 ]
    eval                : temporary : [ -64 .. 64 ]
}


main =          irreversible,
                assign (flipped_something = 0),
                try new_mark else try [
                    assert (!passed),
                    assign (passed = 1),
                ]
                else end_game.

new_mark =      find empty field,
                replace by mark,
                flip_rows,
                assign (passed = 0).

flip_rows =     all directions try flip_row,
                assert (flipped_something).

flip_row =      step,
                points at opponent's mark,
                replace by own mark,
                do step while [
                    points at opponent's mark,
                    replace by own mark
                ],
                points at own mark,
                assign (flipped_something = 1).

end_game =      assign (eval = count(own mark) - count(opponent's mark)),
                try [ assert (eval > 0), win, ]
                else try [ assert (eval < 0), loss, ]
                else draw.
```

A.5 Checkers

```
dimensions (8,8)

layout
{
    .*.*.*.*
    *.*.*.*.
    .*.*.*.*
    *.*.*.*.
    .*.*.*.*
    *.*.*.*.
    .*.*.*.*
    *.*.*.*.
}

pieces
{
    man                 'w' 'b'
    king                'W' 'B'
}


main =                  try either king_capture or man_capture
                        else try either king_move or man_move
                        else loss.

man_move =              irreversible,
                        find man,
                        pick up,
                        either northwest or northeast,
                        step,
                        points at empty field,
                        put down,
                        try promotion.

man_capture =           irreversible,
                        find man,
                        pick up,
                        man_capture_step,
                        put down,
                        try promotion.

man_capture_step =      either northwest or northeast,
                        step,
                        points at opponent's piece,
                        replace by empty field,
                        step,
                        points at empty field,
                        try man_capture_step.
```

```
promotion =             assert (row == 8),
                        replace by king.

king_move =             find king,
                        pick up,
                        diagonal,
                        step,
                        points at empty field,
                        put down.

king_capture =          irreversible,
                        find king,
                        pick up,
                        king_capture_step,
                        put down.

king_capture_step =     diagonal,
                        step,
                        points at opponent's piece,
                        replace by empty field,
                        step,
                        points at empty field,
                        try king_capture_step.
```

A.6 Chess

```
# This implementation does not check the 50–move rule !

dimensions (8,8)

pieces
{
    pawn            'P' 'p'
    rook            'R' 'r'
    knight          'N' 'n'
    bishop          'B' 'b'
    queen           'Q' 'q'
    king            'K' 'k'
}

properties
{
     may_castle_left, may_castle_right        : player : [ 0 .. 1 ]

     en_passant_column                        : board  : [ 0 .. 8 ]
     # 0      – no en passant pawn
     # 1 .. 8 – pawn at (en_passant_column,4) may be captured en passant
}
```

```
main =              try legal_move
                    else try [ find own king, test attacked, loss ]
                    else draw.                  # stalemate

legal_move =        either pawn_move or rook_move or knight_move or
                           bishop_move or queen_move or king_move or
                           castle_move,
                    try
                        [ not find opponent's king, win ]
                    else
                        test [ find own king, not attacked ].

pawn_move =         irreversible,
                    find own pawn,
                    pick up,
                    either pawn_forward_move or pawn_capture_move,
                    put down,
                    try promotion.

pawn_forward_move = north,
                    step,
                    points at empty field,
                    either [                    # double forward
                        assert (row == 3),
                        step,
                        points at empty field,
                        assign (en_passant_column = 9 - column),
                    ]
                    or assign (en_passant_column = 0).

pawn_capture_move = either northwest or northeast,
                    step,
                    either points at opponent's piece or[
                        points at empty field, # en passant capture
                        assert (en_passant_column == column),
                        south,
                        step,
                        points at opponent's pawn,
                        replace by empty field,
                        step backward,
                    ],
                    assign (en_passant_column = 0).

promotion =         assert (row == 8),
                    either replace by queen or replace by knight or
                           replace by bishop or replace by rook.
```

```
rook_move =         find own rook,
                    try [
                        assert (row == 1 && column == 1 &&
                                may_castle_left),
                        irreversible,
                        assign (may_castle_left = 0),
                    ] else try [
                        assert (row == 1 && column == 8 &&
                                 may_castle_right),
                        irreversible,
                        assign (may_castle_right = 0),
                    ],
                    pick up,
                    orthogonal,
                    repeat 0 .. infinity times [
                        step,
                        points at empty field,
                    ],
                    step,
                    not points at own piece,
                    put down,
                    assign (en_passant_column = 0).

bishop_move =       find own bishop,
                    pick up,
                    diagonal,
                    repeat 0 .. infinity times [
                        step,
                        points at empty field,
                    ],
                    step,
                    not points at own piece,
                    put down,
                    assign (en_passant_column = 0).

queen_move =        find own queen,
                    pick up,
                    any direction,
                    repeat 0 .. infinity times [
                        step,
                        points at empty field,
                    ],
                    step,
                    not points at own piece,
                    put down,
                    assign (en_passant_column = 0).
```

```
king_move =         find own king,
                    pick up,
                    any direction,
                    step,
                    not points at own piece,
                    put down,
                    try [
                        assert (may_castle_left || may_castle_right),
                        irreversible,
                        assign (may_castle_left = may_castle_right = 0),
                    ],
                    assign (en_passant_column = 0).

knight_move =       find own knight,
                    pick up,
                    orthogonal,
                    step,
                    either turn 45 or turn -45,
                    step,
                    not points at own piece,
                    put down,
                    assign (en_passant_column = 0).

castle_move =       irreversible,
                    either
                        [ assert (may_castle_left), west ]
                    or
                        [ assert (may_castle_right), east ],
                    # find king,
                    try
                        [ white to move, move to (5,1) ]
                    else
                        move to (4,1),
                    test [
                        step,
                        do [
                            points at empty field,
                            step,
                        ] while not points at own rook
                    ],
                    test repeat 3 times [ not attacked, step ],
                    assign (may_castle_left = may_castle_right = 0),
                    pick up, # the king
                    repeat 2 times step,
                    put down, # the king
                    while step do nothing
                    pick up, # the rook
                    do step backward while not points at own king,
                    step backward,
                    put down, # the rook
                    assign (en_passant_column = 0).
```

```
attacked =          test either atckd_by_pawn or atckd_by_rook or
                                atckd_by_knight or atckd_by_bishop or
                                atckd_by_queen or atckd_by_king.

atckd_by_pawn =     either northwest or northeast;
                    step,
                    points at opponent's pawn.

atckd_by_rook =     orthogonal,
                    do step while points at empty field,
                    points at opponent's rook.

atckd_by_bishop =   diagonal,
                    do step while points at empty field,
                    points at opponent's bishop.

atckd_by_queen =    any direction,
                    do step while points at empty field,
                    points at opponent's queen.

atckd_by_king =     any direction,
                    step,
                    points at opponent's king.

atckd_by_knight =   orthogonal,
                    step,
                    either turn 45 or turn -45,
                    step,
                    points at opponent's knight.
```

References

[1] S. Papert. *Mindstorms: Children, Computers, and Powerful Ideas*. Harvester Wheatsheaf, New York, N.Y., second edition, 1993.

[2] J.W. Romein. *Multigame — An Environment for Distributed Game-Tree Search*. PhD thesis, Faculty of Sciences, Department of Mathematics and Computer Science, Vrije Universiteit, Amsterdam, the Netherlands, To appear.