INTERNATIONAL COMPUTER SCIENCE INSTITUTE

1947 Center St. • Suite 600 • Berkeley, California 94704-1198 • (510) 643-9153 • FAX (510) 643-7684

# Sather 1.0 Tutorial

Michael Philippsen*
phlipp@icsi.berkeley.edu

TR-94-062

Version 0.1, December 1994

**Abstract**

This document provides basic information on how to obtain your copy of the Sather 1.0 system and gives several pointers to articles discussing Sather 1.0 in more detail.

We thoroughly describe the implementation of a basic chess program. By carefully reading this document and the discussed example program, you will learn enough about Sather 1.0 to start programming in Sather 1.0 yourself. This document is intended for programmers familiar with object oriented languages such as Eiffel or C++. General information on object oriented programming can be found in [5].

The main features of Sather 1.0 are explained in detail: we cover the difference between subtyping and implementation inheritance and explain the implementation and usage of iters. Moreover, the example program introduces all the class elements (constants, shared and object attributes, routines and iters) are introduced. Most statements and most expressions are also discussed. Where appropriate, the usage of some basic features which are provided by the Sather 1.0 libraries are demonstrated. The Tutorial is completed by showing how an external class can be used to interface to a C program.

*On leave from Department of Computer Science, University of Karlsruhe, Germany

# Contents

# 1 About Sather 1.0

Sather is an object oriented language which aims to be simple, efficient, safe, and non-proprietary. One way of placing it in the "space of languages" is to say that it aims to be as efficient as C, C++, or Fortran, as elegant and safe as Eiffel or CLU, and support higher-order functions and iteration abstraction as well as Common Lisp, Scheme, or Smalltalk.

Sather has parameterized classes, object-oriented dispatch, statically-checked strong (contravariant) typing, separate implementation and type inheritance, multiple inheritance, garbage collection, iteration abstraction, higher-order routines and iters, exception handling, assertions, preconditions, postconditions, and class invariants. Sather programs can be compiled into portable C code and can efficiently link with C object files. Sather has a very unrestrictive license which allows its use in proprietary projects but encourages contribution to the public library.

## 1.1 Where can I find Sather?

Information on Sather can be found on the Mosaic page `http://www.icsi.berkeley.edu/Sather`. From that page, you can reach various documents related to Sather. There also is a list of frequently asked questions. Another source of information is the newsgroup `comp.lang.sather` that is devoted to discussion of Sather related issues.

There is a Sather mailing list maintained at the International Computer Science Institute (ICSI). Since the formation of the newsgroup, this list is primarily used for announcements. To be added to or deleted from the Sather list, send a message to `sather-request@icsi.berkeley.edu`.

If you have problems with Sather or if you want to discuss Sather related questions that are not of general interest, mail to `sather-bugs@icsi.berkeley.edu`. This is also where to send bug reports and suggestions for improvements.

The current ICSI Sather 1.0 compiler, the manual, this tutorial, and the Sather FAQ can be obtained by anonymous ftp from

```
ftp.icsi.berkeley.edu        /pub/sather
```

The distribution file is called `Sather-1.0.*.tar.Z`. The wildcard is to be replaced by the number of the latest release. At the time this tutorial was written three sites have mirrored the Sather distribution:

```
ftp.sterling.com             /programming/languages/sather
ftp.uni-muenster.de          /pub/languages/sather
maekong.ohm.york.ac.uk       /pub/csp
```

## 1.2 Where can I read about Sather?

There are various papers on Sather 1.0, on earlier versions, primarily on Sather 0.5 which is somewhat different, and on pSather which is a parallel extension of Sather.

Most of the papers listed here are directly available from the Mosaic page mentioned above. Others can be retrieved via anonymous ftp from `ftp.icsi.berkeley.edu` under `/pub/techreports`. As a last resort, hardcopies may be ordered for a small fee. Send mail to `info@icsi.berkeley.edu` for more information.

The current language specification is published in [10]. This document can be found next to the code on the ftp server mentioned above. Obviously the file is called `manual.ps`.

Sather's general design and the differences from Eiffel have been presented in [6, 7, 8, 9]. The type system is presented in depth in [13]. Moreover, ICSI technical papers report on other specific issues, see [2, 4, 11, 13].

Sather has been analyzed from an external point of view. Comments and comparisons can be found in [1, 3, 12].

### 1.3 Related Work: Sather-K

*Although we know a lot about Sather-K, which is being developed in Karlsruhe, Germany, it is not yet available online. Future versions of this Technical Report, which can be accessed from anonymous ftp will have some more details.*

### 1.4 Planned Changes to this Tutorial

Currently Sather Tutorial Chess does not use the file I/O libraries of Sather 1.0. Since it takes some time to get used to these libraries, the Tutorial definitively should explain them.

Hence, later versions of this Technical Report, which can be accessed from anonymous ftp will be extended in that respect. We will either introduce a way to save the current state of a game and resume at a later program invocation. Or we will supply a library of standard openings and use that information when generating automatic moves.

# 2 Sather Tutorial Chess

Sather Tutorial Chess is *not* an expert chess program. In fact, it is quite easy to win against the computer. Moreover, the implementation is very inefficient in certain parts of the code. The idea is to simply provide a context for demonstrating and explaining various features of Sather and not to show a world class chess program.

To make the best use of this tutorial, the Sather 1.0 system should be properly installed and the following files should be available online:

**hello.sa** This file contains is the standard *Hello World* program. It does not belong to Sather Tutorial Chess but is included as an initial exercise.

**Makefile** This is the Makefile for Sather Tutorial Chess.

**SChess.sa** This is the main Sather file.

**XInterf.sa** This is an additional Sather file. Although the code could have been in SChess.sa, it is kept in a separate file for explanatory reasons.

**DefaultA.sa** If your system is not running the X11 window system, this file is used for compilation and linking.

**DefaultX.sa** Otherwise, this file is used instead.

**XCW.c** This C file provides the interface to the X11 window system. If you do not use X11, the Makefile will detect this and generate an executable that does not depend on or use XCW.c.

**bitmaps** This directory has bitmaps for all the chess pieces which are used in XCW.c.

## 2.1 Hello World Program

The file `hello.sa` is the standard *Hello World* program. Sather programs usually have file names with the extension `.sa`. To compile it, simply enter `cs hello.sa`. The command for invoking the compiler is easy to remember, since `cs` stands for "Compile Sather". After successful compilation you can execute it by entering `a.out`. If the current directory is not in your search path, enter `./a.out`.

Only proceed after having successfully compiled and executed the *Hello World* program. If something went wrong, check your installation of the Sather 1.0 system. The file `Doc/Installation` might be helpful for diagnosing problems.

```
-- This is the standard Hello World program
-- implemented in Sather 1.0
class MAIN is
  main is
    #OUT + "Hello World\n";
  end;
end;
```

The first two lines of the file are comments. Comments start with two minus signs. The comment cannot be explicitly closed, they end at the end of the line. The class MAIN has a special purpose in Sather. Unless altered by compiler flags, the routine **main** of MAIN is started when a compiled Sather program is invoked by the user. In **main** there is only one statement. This statement is responsible for several things: At first #OUT creates a new object of class OUT. Class OUT is a basic class provided by Sather. In the implementation of class OUT which can be found in the library file Library/out.sa there are several routines that can be invoked on an object of that class. One of these routines has the signature

plus(s:STR);

Make sure that you look at the library file Library/out.sa and find the routine used in the Hello World program. It is necessary for using the Sather 1.0 system that you are familiar with the libraries and the routines provided by them. The routine plus takes one string argument and "adds" this argument to the object before returning the modified object. In line 5 of the program the routine *plus* is called implicitly, by the operator + which itself is syntactic sugar for the call of plus.

In Sather 1.0 a string is enclosed in double quotes ("). Similar to C, `\n` stands for the carriage return/line feed.

## 2.2 Getting Started

The other files mentioned above are needed for Sather Tutorial Chess. They could be derived from this document by extracting and concatenating the code segments explained in the remainder. Unless otherwise noted, the code segments go to the file `SChess.sa`.

For the presentation, code segments are numbered on the right of the code. Numbering is restarted with line 1 either when a new Sather code file is started or with the beginning of a new section.

You can create an executable Sather Tutorial Chess program by invoking the compiler. This is done by staring the execution of the Makefile:

```
make
```

The Makefile finds out whether your system runs then X Windows. Depending on the result, the appropriate Sather code files are compiled and linked together. The executable is called

```
SChess
```

After invoking Sather Tutorial Chess, you are the white player. The computer is responsible for the moves of black. Later, in section 3.2 we will show how this default behavior can be changed.

## 2.3 Class Hierarchy of Sather Tutorial Chess

Let us first discuss the basic design decisions that led to our implementation of Sather Tutorial Chess. The central object is the *board*. The board knows about its state, which is – roughly speaking – the set of *pieces*, and is capable of applying *moves* to itself. Moves and pieces are other types of objects. A "moves" knows about the *piece* that is moved and knows both the starting and the final position of the move. Pieces and moves use *position* objects to represent the position on the board.

Besides those objects that are used for representing and handling the chess game, there are several helper objects that are necessary for interfacing with the user. For both players there is a player object. This player objects hides the origin of a move from the chess engine. The player object is asked to return a move. This call is either forwarded to the user or to the searching strategy of the computer player. Hence, the same chess engine can be used for all four possible pairings of human and automatic players.

Another object is used for handling the display of the chess board. If required, this interface can ask the user to enter a move in standard chess notation. The implementation provides both a plain ASCII interface and an interface to the X Window system.

The description will start with the class MAIN which contains the basic loop of the game. In section 4 we discuss the display objects. After that, section 5 deals with the players. Then the other classes are presented in the following order: move in section 6, position in section 7, board in section 8 and finally pieces in section 9.

# 3 Class Main

The class MAIN has a special purpose in Sather. Unless altered by compiler flags, the routine **main** of MAIN is started when a compiled Sather program is invoked by the user. Class names must be in capital letters.

Although it is possible, it is unusual to create objects of class MAIN. Therefore, attributes should be declared *shared*. Shared attributes of a class exist and can be accessed even if no objects are created. Above that, shared attributes are globally accessible by all objects of a given type.

Here we declare shared variables that can hold pointers to the chess board, the display object, and to the players. The variable board can hold an object of type BOARD, which is specified by the implementation of *class* BOARD, see section 8 for details. The other four variables can hold objects of the *abstract type* $CHESS_DISPLAY or $PLAYER, respectively. These objects can be created by classes that are explicitly declared to be subtypes of the abstract types. The difference between classes and abstract types that is visible here by the use of the $ symbol in the type identifiers and will be explained in more detail in section 4.

```
class MAIN is
  shared board : BOARD;
  shared display : $CHESS_DISPLAY;
  shared white, black, player : $PLAYER;
```

This is a good point to introduce Sather's ubiquitous basic data types. Upon declaration of basic types, these are initialized automatically.

- BOOL defines value objects which represent boolean values. The initial value is false.
- CHAR defines value objects which represent characters. The initial value is '\0'.
- STR defines reference objects which represent strings.
- INT defines value objects which represent machine-dependent integers. The size is implementation dependent but must be at least 32 bits. The two's complement representation is used to represent negative values. Bit operations are supported in addition to numerical operations.
- INTI defines reference objects which represent infinite precision integers.
- FLT, FLTD, FLTX, and FLTDX define value objects which represent floating point values according to the single, double, extended, and double extended representations defined by the IEEE-754-1985 standard.
- FLTI defines reference objects which represent arbitrary precision floating point objects.
- The parameterized type ARRAY{T} defines general purpose array objects of type T. For example, ARRAY{STR} represents an array whose elements are strings of type STR.
- TUP names a set of parameterized value types called "tuples", one for each number of parameters. Each has as many attributes as parameters and they are named "t1", "t2", etc. Each is declared by the type of the corresponding parameter (e.g. TUP{INT,FLT} has attributes t1:INT and t2:FLT). It defines a create routine with an argument corresponding to each attribute.

There are more basic data types. Since these are irrelevant for this Tutorial, the interested reader is referred to the manual [10].

Sather distinguishes between reference objects and value objects. (Other types of objects are not mentioned in this tutorial.) Experienced C programmers immediately catch the difference when told about the internal representation: Value types are C `struct`s and reference types are pointers to `struct`s.[1] Because of that difference, reference objects can be referred to from more than one variable. Value objects can not. The basic types mentioned above (except arrays) are value classes. Reference objects must be explicitly allocated with **new**. Variables have the value void until an object is assigned to them. Void for reference objects is similar to a void pointer in C. Void for value objects means that a predefined value is assigned (0 for INT*, \0 for CHAR, false for BOOL, 0.0 for FLT*). Accessing a void value object will always work. Accessing a void reference object usually will be a fatal error.

There are some more differences between value types and reference types but they are beyond the scope of this tutorial[2].

## 3.1 Routine main

The routine **main** of MAIN is started when Sather Tutorial Chess is invoked. Similar to C, the parameter args returns the command line which is used to invoke the program. If **main** is declared without parameters, the command line and any arguments are ignored. Since the routine **main** is declared to return an integer, this will specify the exit code of the program when it finishes execution. If **main** is declared without return parameter, no exit code will be returned.

```
5  main(args:ARRAY{STR}):INT is
6    if ~setup(args) then -- ~ is the boolean NOT
7      -- If the given command line arguments are not acceptable, setup
8      -- returns false. Then the program terminates and returns -1.
9      return -1;
10   end;
```

After invocation, the routine setup analyzes the given command line arguments. It returns true if the given parameters are acceptable and false otherwise. If acceptable, setup has some side effects: it creates objects for the players, for display, and for board. Later on these objects are accessible via the variables declared in lines 2–4.

If setup had returned true, the board, the display, and the players have been created when execution reaches line 11 where the game starts. The game is essentially a loop (lines 11–32) in which the current player is asked to enter/generate a move. The result is then assigned to the implicitly declared local variable move (line 12). The type of move is derived from the return type of player.getmove because of "::=". The type could also have been specified explicitly as follows:

```
move : MOVE := player.getmove(board);
```

Another way could be to declare the variable first and then assign in a second statement:

```
move : MOVE;
move := player.getmove(board);
```

The scope of move is defined by the surrounding block, i.e., the loop statement.

Later we will find out that player.getmove is a dispatched call. But let's skip this for now.

[1] Furthermore, you are not allowed to have pointers directly to fields of `struct`s.

[2] Some other difference are named here because of completeness:

- Value type must inherit from AVAL{T} instead of AREF{T}.
- The writer routine takes different forms for reference and value types. For reference types, it takes a single argument whose type is the attribute's type and has no return value. Its effect is to modify the object by setting the value of the attribute. For value types, it takes a single argument whose type is the attribute's type, and returns a copy of the object with the attribute set to the specified new value, and whose type is the type of the object. This difference arises because it is not possible to modify value objects once they are constructed. Study the complex number library in file Library/cpx.sa.

The loop is terminated if the move is a quit. The test occurs in line 13 in the **until!** expression, which is a call to a special iter: each time **until!** is called, the given boolean expression is evaluated. If false, **until!** "quits" which breaks the immediately surrounding loop, i.e., terminates the game.

If the program flow reaches the statement after **until!** the latter did not terminate the loop. Since some move has been returned from player.getmove it must be checked and applied to the board. This is done in line 14 by the routine check_n_apply_move which returns false if the move could not be applied properly.

After application of the move to the board in line 15, the display object is called to update the view of the board.

Later we will find out that the calls to display.update in line 15, to display.king_check() in line 25, to display.invalid_move in line 30, and to display.close in line 35 all are dispatched calls. But again, let's skip this for now.

```
11    loop
12      move ::= player.getmove(board);
13      until!(move.isquit);
14      if board.check_n_apply_move(move) then
15        display.update(board.str);
16        -- Set player to the next player
17        if board.white_to_play then
18          player := white;
19        else
20          player := black;
21        end;
22        -- Find out whether the king of the current player is in
23        -- check. If so, have the display talk about the situation.
24        if board.my_king_isin_check then
25          display.king_check(board.white_to_play);
26        end
27      else
28        -- The move was invalid. Display this. By not changing
29        -- the current player, the same player is asked to try again.
30        display.invalid_move;
31      end;
32    end; -- of loop
33    -- The game is over, since the current player issued a "quit-move".
34    -- Close the display.
35    display.close;
36    return 0;
37  end;
```

## 3.2 Routine setup

This setup routine gets the command line arguments and returns a BOOL. The return value of setup is true, iff the parameters have been acceptable.

To start Sather Tutorial Chess use:

```
SChess  [<white> <black>] [<Displ>]
         <white> can be either H for Human Player
                            or     C for Computer Player
```

```
              <black> dito
              <Displ> can be either X for X Interface
                                 or     A for ASCII Terminal
        The default behavior is SChess H C X
```

The type of the args parameter, ARRAY{STR}, is an instantiation of the parameterized basic type ARRAY{T}. The source code can be found in file Library/array.sa. An c of type ARRAY{T} stores elements of type T. If c is not void, the first element can be accessed by c[0]. c.size returns the number of elements stored in the array. c[c.size-1] accesses the last element.

```
38 setup(args:ARRAY{STR}):BOOL is
39   -- set defaults
40   ret  : BOOL := true; -- the default is that the parameters are ok
41   p    ::= #ARRAY{CHAR}(2);
42   p[0] := 'H';         -- default: human player
43   p[1] := 'C';         -- default: computer player
44   d    : CHAR := 'X';  -- type of display
```

First of all, setup creates a few variables that will hold the result of the evaluation of the command line arguments. A novelty is in line 41, where p is declared to be a character array and space is allocated for it. The array is created and initialized by calling the **create** routine of the class ARRAY. The # symbols is syntactic sugar for calls of **create** routines. If the **create** routine need additional arguments, they must be supplied behind the # symbol. Here the array has two characters which can be accessed as p[0] and p[1].

In the following code segment, the arguments get processed in a loop (lines 47–65). The first argument, args[0] is left out, since this contains the name of the running program. Here, loop termination is implemented in line 47 by the use of the iter upto! which is declared in the INT library. (The INT class is implemented in the file Library/int.sa.) The iter upto! returns an integer value each time it is called. Here the first call will return 1, the argument specifies the upper bound. In the second call upto! will return 2, then 3, ..., and finally args.size-1. The next call will quit the iter and terminate the immediately surrounding loop, i.e., program execution will continue in line 72.

For analysis of single parameters we use routines, provided by the STR class. The string class, which is implemented in the file Library/str.sa offers a routine char(int) that returns the character with the specified number. Since strings are arrays of characters, the first character of a string can be accessed by char(0). The character class which is implemented in the file Library/char.sa has routines upper and lower that return an upper case or lower case version of the character they are called upon. The routine head(k) returns the first k characters of a string.

```
45   if args.size > 1 and args.size <= 4 then
46     player_cnt : INT := 0;
47     loop i::=1.upto!(args.size-1);
48       if args[i].size >= 4 and args[i].head(4).lower="help" then
49         ret := false;
50       end;
51       tmp : CHAR := args[i].char(0).upper;
52       case tmp
53       when 'A', 'X' then   -- ASCII- or X-Display if available
54         d := tmp;
55       when 'H', 'C' then   -- Human or Computer player
56         if player_cnt < 2 then
```

```
57      p[player_cnt] := tmp;
58      player_cnt := player_cnt + 1;
59    else
60      ret := false;
61    end;
62  else
63    ret := false;
64  end;
65 end; -- of loop
66 elsif args.size /= 1 then   -- not equal
67  -- The parameters are not acceptable.
68  ret := false;
69 else
70  -- use defaults. The else could have been omitted.
71 end;
```

Boolean expressions are evaluated with short-circuit semantics. For an and this means that the second operand is only evaluated if the first operand was true. For an or the second operand is evaluated if the first one was false. Lines 45 and 48 are good examples.

Sather's **case** statement (lines 52–64) is used for processing the command line parameters other than "help". The variable tmp is evaluated and depending on the result, the first matching **when** branch is taken. Note, that multiple expressions can be given for comparison in each branch.

Depending on the analysis of the command line arguments either all global objects needed for the chess program are created in lines 79–88 or the user is informed about the correct parameter syntax in lines 90–96. The Output class OUT is defined in file Library/out.sa. The idea of using the class is to create an output object and "add" the things that should be output to this object. The plus is overloaded so that all basic types can be output in this fashion. As usual, `\n` indicates a carriage return/line feed.

```
72 if ret then
73  display := DEFAULT::display(d); -- Creates Display object. Described below.
74  board   := #BOARD;
75  if p[0] = 'H' then
76    -- An object of type HUMAN is created. In contrast to BOARD,
77    -- this object has a special create routine, that needs an argument.
78    white   := #HUMAN(board.white_to_play);
79  else
80    white   := #MINMAX(board.white_to_play);
81  end;
82  if p[1] = 'H' then
83    black   := #HUMAN(~board.white_to_play);
84  else
85    black   := #MINMAX(~board.white_to_play);
86  end;
87  -- the first player is White
88  player  := white;
89 else
90  #OUT+"To start Sather Tutorial Chess use: \n";
91  #OUT+"args[0] [<white> <black>] [<Displ>]\n";
92  #OUT+"        <white> can be either H for Human Player\n";
```

```
    #OUT+"                        or     C for Computer Player\n";
    #OUT+"          <black> dito\n";
    #OUT+"          <Displ> can be either X for X Interface\n";
    #OUT+"                        or     A for ASCII Terminal\n";
  end;
  -- Since setup has a return parameter, a result
  -- has to be returned to the caller.
  return ret;
 end;  -- of setup
end; -- of class MAIN
```

# 4 Type $CHESS_DISPLAY and Related Classes

## 4.1 Type $CHESS_DISPLAY

Sather differentiates between concrete types and abstract types. In Sather each object has a unique concrete type that determines the operations that may be performed on it. Classes define concrete types and give implementations for the operations. Abstract types however, only specify a set of operations without providing an implementation. This set of operations is called the interface of the type. An abstract type corresponds to a set of concrete types which obey that interface.

$CHESS_DISPLAY is an abstract type. Names of abstract types must be in capital letters. The leading $ differentiates abstract from concrete types.

In the body of the type declaration (lines 2–14), the operations are given without any implementation. Formal parameters must have names. However, since these are not used, the names serve only documentary purposes.

For example, consider the case where you want to have a simple integer variable in all concrete types/classes that are subtypes of an abstract type. An integer attribute a has two implicit routines, a reader which has the signature a:INT and a writer with the signature a(new_value:INT). Since the abstract type hides implementation details from the interface, one has to put both signatures in the body of the type. This gives room for changing the implementation of a in the classes. (In the abstract type below, there are however no attributes.)

```
1  type $CHESS_DISPLAY is
2    -- Display the state of the board
3    redraw(board:ARRAY{CHAR});
4    update(board:ARRAY{CHAR});
5    showmove(text:STR);
6    -- Inform player about certain conditions
7    invalid_move;
8    thinking(white_to_move:BOOL);
9    king_check(white_to_move:BOOL);
10   -- Interact with the player
11   getmove(white_to_move:BOOL):MOVE;
12   ask_pawn_xchg:CHAR;
13   -- Close
14   close;
15 end; -- of abstract type $CHESS_DISPLAY
```

The string interface (ARRAY{CHAR}) to board needs some explanation: The board is represented by 64 characters. Each character specifies the piece on a particular position of the board.

| | | | |
|---|---|---|---|
| ' ' | no piece | 'P' | Pawn |
| 'B' | Bishop | 'Q' | Queen |
| 'K' | King | 'R' | Rook |
| 'N' | Knight | | |

Capital characters represent white pieces, small characters stand for black pieces. The first character in board specifies board position "a1", the last "h8".

All concrete classes that are subtype of $CHESS_DISPLAY must at least have all the above routines (or implicitly declared routines.)

## 4.2 Class CHESS_DISPLAY

This is a concrete type or class which is a subtype of $CHESS_DISPLAY. The subtype relation is expressed by the < symbol in line 16. This concrete class however will *not* be used to instantiate objects, i.e., there will be no objects of type CHESS_DISPLAY. The main purpose of this class is to declare attributes and routines that are common to other classes of type $CHESS_DISPLAY, which **include** the implementation of this class. Hence, whereas $CHESS_DISPLAY is used to express the subtype relation, the class CHESS_DISPLAY is used for code inheritance.

The first two routines are included unchanged in ASCII_DISPLAY and replaced in X_DISPLAY.

A **create** routine has to be provided if objects of that concrete type are created. SAME denotes the type of the class in which it appears. As explained in ASCII_DISPLAY below, it is a good idea to return SAME instead of CHESS_DISPLAY, if the create routine is meant to be included.

The expression **new** is used in line 18 to allocate space for (reference) objects (and may only appear in reference classes.) New returns a (reference) object of type SAME. All attributes and array elements are initialized to void.

```
16 class CHESS_DISPLAY < $CHESS_DISPLAY is
17  create:SAME is
18   return new;
19  end;
20  update(board:ARRAY{CHAR}) is
21   redraw(board);
22  end;
```

The following two routines do not provide a basic implementation. However, for consistency with the interface required by $CHESS_DISPLAY, they have to exist. When the code of class CHESS_DISPLAY is included, special implementations of redraw and getmove must be provided that replace the dummies given here.

To make sure that these implementations of redraw and getmove are not called erroneously, an exception is raised by the **raise** statement (lines 24 and 27). Since redraw does not have a return parameter, the body of the routine could have been empty. In getmove either a **return** or a **raise** is required because getmove has a return parameter.

```
23  redraw(board:ARRAY{CHAR}) is
24   raise "INTERFACE: invalid call to redraw\n";
25  end;
26  getmove(white_to_move:BOOL):MOVE is
27   raise "INTERFACE: invalid call to getmove\n";
28  end;
```

The following four routines provide code that is meant to be included unchanged in other implementations of classes that are subtypes of $CHESS_DISPLAY. Each of the four routines makes use of a private routine showtext which is not completely coded here. Classes that **include** the implementation of CHESS_DISPLAY must provide complete implementations of showtext.

```
29  invalid_move is
30   text : STR;
31   text := "ERROR: Invalid move....try again";
32   showtext(text);
33  end;
```

```
34 thinking(white_to_move:BOOL) is
35   text : STR;
36   if white_to_move then
37     text := "White";
38   else
39     text := "Black";
40   end;
41   text := text + " is thinking ... please wait ...";
42   showtext(text);
43 end; -- of thinking
44 king_check(white_to_move:BOOL) is
45   text : STR;
46   if white_to_move then
47     text := "--> White";
48   else
49     text := "--> Black";
50   end;
51   text := text + " is in check!";
52   showtext(text);
53 end; -- of king_check
54 showmove(text:STR) is
55   showtext(text);
56 end;
```

A routine declared **private** can only be called from code that is in the same class as the routine.

```
57 private showtext(text:STR) is
58   -- Optional protection against implementation errors
59   raise "INTERFACE: invalid call to showtext\n";
60 end;
```

The following routine ask_pawn_xchg is included in both ASCII_DISPLAY and X_DISPLAY without change. The loop (line 64–72) is *not* terminated by means of an iter. Instead, the termination is done by the **return** statement in line 69.

In line 66 is an example of user input. The class IN is specified in the file Library/in.sa. Among others, IN provides a routine get_str that accepts a string input from the use via the standard I/O-device. Calls like CLASS::<routine> do not refer to a particular object of the class but call the routine on a void object.

```
61 ask_pawn_xchg:CHAR is
62   newpiece : STR;
63   ret : CHAR;
64   loop
65     #OUT+"Do you prefer a QUEEN or a KNIGHT?\n";
66     newpiece := IN::get_str.upper;
67     ret := newpiece.char(0);
68     if ret = 'Q' or ret = 'K' then
69       return ret;
70     end;
71     #OUT+"Please enter QUEEN or KNIGHT.\n"
```

```
72     end;
73   end; -- of ask_pawn_xchg
74   -- The following routine is included unchanged in ASCII_DISPLAY
75   -- and replaced in X_DISPLAY.
76   close is
77   end;
78 end; -- of CHESS_DISPLAY
```

### 4.3 Class ASCII_DISPLAY

This concrete class is a subtype of $CHESS_DISPLAY. It provides an implementation for at least the signatures given in the specification of $CHESS_DISPLAY.

ASCII_DISPLAY inherits the implementation of class CHESS_DISPLAY by the **include** statement. The **include** statement is semantically equivalent to the following editor operation: replace the **include** statement by the implementation code of the included class. (Includes have to be resolved recursively.)

Without code duplication, ASCII_DISPLAY inherits the implementation of the following routines, at the **include** statement.

```
create:SAME
redraw(board:ARRAY{CHAR})  --> is replaced below
update(board:ARRAY{CHAR})
getmove(white_to_move:BOOL):MOVE  --> is replaced below
invalid_move
thinking(white_to_move:BOOL)
king_check(white_to_move:BOOL)
showmove(text:STR)
private showtext  --> is replaced below
ask_pawn_xchg:CHAR
close
```

Only the routines marked with "–>" are replaced by a specific implementation. To make the idea of textual inclusion even more understandable consider the included version of **create**.

```
create:SAME;
```

Although originally written in CHESS_DISPLAY, the routine **create** does not return an object of type CHESS_DISPLAY after being included in ASCII_DISPLAY. Instead, **create** returns an object of type ASCII_DISPLAY.

```
79 class ASCII_DISPLAY < $CHESS_DISPLAY is
80   include CHESS_DISPLAY;
```

Redrawing the board on the ASCII_DISPLAY is an excellent example of two nested loops, both of which are governed by iters (lines 88–91 and lines 87–89).

The iter downto! in line 85 is another iter from the INT class, which can be found in file Library/int.sa. As expected, 7.downto(0) iteratively returns the integer value 7, 6, 5, ..., 0 and with the next call terminates the surrounding loop, i.e., the loop from line 85 to line 91.

The iter step! in line 87 is just another iter the INT class provides. Beginning at the integer it is called upon, it will return as many integers as indicated by its first argument. The difference between two subsequent return values is given by the second argument. If step! is called for the ninth time, it will quit and immediately terminate the surrounding loop (line 87–89). Note, that for the two nested loops, only the innermost loop is terminated.

```
81 redraw(board:ARRAY{CHAR}) is
82   #OUT+"The current board: (small characters = black pieces)\n";
83   #OUT+"    a  b  c  d  e  f  g  h \n";
84   #OUT+"   ------------------------\n";
85   loop i::=7.downto!(0);
86     #OUT+(i+1)+"|";
87     loop j::=(8*i).step!(8,1);
88       #OUT+" "+board[j]+" "
89     end;
90     #OUT+"|"+(i+1)+"\n";
91   end;
92   #OUT+"   ------------------------\n";
93   #OUT+"    a  b  c  d  e  f  g  h \n";
94 end; -- of redraw
```

The following OUT::flush in line 106 tells the OUT class, that all characters that are buffered should be output immediately. Normally, the buffer is only flushed, if a \n is seen in the character stream.

```
95  getmove(white_to_move:BOOL):MOVE is
96    move : MOVE;
97    move_str : STR;
98    loop
99      #OUT+"Please enter a move for";
100     if white_to_move then
101       #OUT+" white: ";
102     else
103       #OUT+" black: ";
104     end;
105     #OUT+"(e.g. d2-d3 or help) ";
106     OUT::flush;
107     move_str := IN::get_str.lower;
108     -- The string class provides a routine head(x), which returns the first
109     -- x characters of a string.
110     if move_str.size >= 4 and move_str.head(4) = "help" then
111       #OUT+"Valid moves are:\n";
112       #OUT+"  ordinary move: d2-d3\n";
113       #OUT+"  king castle  : o-o\n";
114       #OUT+"  queen castle : o-o-o\n";
115       #OUT+"  quit         : quit\n";
116     else
117       move := #MOVE(move_str, white_to_move);
118       -- If the create routine of MOVE could not correctly deal with
119       -- the given move_str move.isok returns false. If a move turns
120       -- out not to be quit or ok, the player is asked to try again.
121       until! (move.isquit or move.isok);
122       #OUT+"ERROR: Invalid syntax....try again\n";
123     end;
124   end;
125   return move;
```

```
  end; -- of getmove
  private showtext(text:STR) is
    #OUT+text+"\n";
  end;
end; -- of ASCII_DISPLAY
```

## 4.4 Class X_DISPLAY

The following code is kept in a separate Sather code file (XInterf.sa). There the class X_DISPLAY is implemented. The implementation is in a different file, to show how spreading of source code across several files works in Sather.

This concrete class is a subtype of $CHESS_DISPLAY. It provides an implementation for at least the signatures given in the specification of $CHESS_DISPLAY.

Due to the **include** statement, X_DISPLAY inherits the implementation of CHESS_DISPLAY in then same way as ASCII_DISPLAY has done before. Without code duplication, X_DISPLAY now has

```
create:SAME  --> is replaced below
redraw(board:ARRAY{CHAR})  -->* is replaced below
update(board:ARRAY{CHAR})  --> is replaced below
getmove(white_to_move:BOOL):MOVE  -->* is replaced below
invalid_move
thinking(white_to_move:BOOL)
king_check(white_to_move:BOOL)
showmove(text:STR)
private showtext  -->* is replaced below
ask_pawn_xchg:CHAR
close  --> is replaced below
```

Only the routines marked with "–>" are replaced by a specific implementation. The arrows marked with * indicate those routines that have been replaced in the ASCII_DISPLAY explained above.

The implementation of X_DISPLAY makes heavy use of the external Chess Window (XCW) implementation. The Sather compiler is informed about the existence of the external routines in the external class XCW which is explained on page 17.

```
class X_DISPLAY < $CHESS_DISPLAY is
  include CHESS_DISPLAY;
  create:SAME is
    XCW::OpenCW("Sather Tutorial Chess");
    return new;
  end;
  redraw(board:ARRAY{CHAR}) is
    XCW::RedrawCW(board);
  end;
  update(board:ARRAY{CHAR}) is
    XCW::UpdateCW(board);
  end;
  showmove(text:STR) is
    XCW::ShowMoveCW(text);
  end;
  private showtext(text:STR) is
    XCW::TextCW(text);
  end;
```

```
19  close is
20    XCW::CloseCW;
21  end;
```

The implementation of getmove is slightly more complicated. The external Chess Window implementation has a routine called GetMoveInCW. This routine has an array of characters as formal parameter. This array is kept in the variable move_chars. To pass the result to the **create** routine of class MOVE in line 36, it must be converted into a string. The latter is stored in the variable move_str.

Several library routines are helpful here. In line 35 routine to_val of class ARRAY{T} is used to set each array element to the given value. The loop in lines 39–41 iteratively adds characters of move_char to the string variable move_str. The iter elt! returns all array elements in order and quits at the end of the array, hence terminating the loop. Note, how elegantly both loop control and work can be combined by use of iters.

```
22  getmove(white_to_move:BOOL):MOVE is
23    text : STR;
24    text := "Please move a";
25    if white_to_move then
26      text := text+" white"
27    else
28      text := text+" black";
29    end;
30    text := text+" piece.";
31    XCW::TextCW(text);
32    move_chars ::= #ARRAY{CHAR}(5); -- create a character array with 5 chars.
33    move_str   ::= #STR;            -- create a string.
34    move       : MOVE;
35    move_chars.to_val(' ');         -- set all 5 chars to ' '
36    XCW::GetMoveInCW(move_chars);
37    -- Construct string out of char array. The iter elt! returns all 5
38    -- characters of move_chars, then quits and terminates the loop.
39    loop
40      move_str := move_str+move_chars.elt!;
41    end;
42    -- Since XCW::GetMoveInCW is guaranteed to return only
43    -- syntactically correct moves, no further plausibility tests
44    -- are required.
45    move := #MOVE(move_str.lower,white_to_move);
46    return move;
47  end; -- of getmove
48 end; -- of X_DISPLAY
```

## 4.5 External Class XCW

XCW provides an X Window interface for chess. The corresponding C code can be found in XCW.c. The routines are used by the implementation of X_DISPLAY.

In this external class definition the interface to routines of XCW.c are specified. The main purpose of this class is to tell the Sather compiler the names and parameters of routines that can be called. The syntax for a call is XCW::<routine_call>.

```
external class XCW is
  OpenCW(title:STR);
  RedrawCW(board:ARRAY{CHAR});
  UpdateCW(board:ARRAY{CHAR});
  GetMoveInCW(move:ARRAY{CHAR});
  ShowMoveCW(move:STR);
  TextCW(text:STR);
  CloseCW;
end;
```

Each external class is typically associated with an object file compiled from a language like C or Fortran. External classes do not support subtyping, implementation inheritance, or overloading. External classes bodies consist of a list of routine definitions. Routines with no body specify the interface for Sather code to call external code. Routines with a body specify the interface for external code to call Sather code.

Each routine name without a body may only appear once in any external class and the corresponding external object file must provide a conforming function definition. Sather code may call these external routines using a class call expression of the form EXT_CLASS::ext_rout(5). External code may refer to an external routine with a body by concatenating the class name, an underscore, and the routine name (e.g., EXT_CLASS_sather_rout).

Only a restricted set of types are allowed for the arguments and return values of these calls. The built-in value types BOOL, CHAR, INT, FLT, FLTD, FLTX, and FLTDX are allowed anywhere and on each machine have the format supported by the C compiler used to compile Sather for that machine.

Moreover, arrays of the above basic types (except BOOL) can be passed as routine parameters. When a Sather program calls such a routine, the external routine is passed a pointer into just the array portion of the object. The external routine may modify the contents of this array portion, but must not store the pointer. There is no guarantee that the pointer will remain valid after the external routine returns.

## 4.6 Class DEFAULT

One of the design decisions of Sather Tutorial Chess has been to provide both an ASCII interface and an interface to the X Window system. To represent that in the code, there are two implementations of a class called DEFAULT. The first implementation which is in the file DefaultX.sa, can handle both an interface to X and to the ASCII terminal:

```
class DEFAULT is
  display(d:CHAR):$CHESS_DISPLAY is
    ret : $CHESS_DISPLAY;
    if d = 'X' then
      -- Create an object of type X_DISPLAY and return it.
      -- To be more specific: # is a short-hand for a call to
      -- the the routine create of type that follows the #.
      ret := #X_DISPLAY;
    else
      ret := #ASCII_DISPLAY;
    end;
    return ret;
```

```
  end;
end;
```

Depending on the value of d either an object of type X_DISPLAY or of type ASCII_DISPLAY is returned to the caller. The call can be found in line 73 of the setup routine of class MAIN, see page 9.

If X is not available, the following implementation which is kept in Sather code file DefaultA.sa, is used instead:

```
class DEFAULT is
  display(d:CHAR):$CHESS_DISPLAY is
    ret : $CHESS_DISPLAY;
    -- Since X is not available, create ASCII-Interface only.
    ret := #ASCII_DISPLAY;
    return ret;
  end;
end;
```

The value of d is ignored here. In either case, an ASCII display is created and returned to the caller. Since no reference to class X_DISPLAY is in the code, the Sather compiler ignores any implementation of that class. The Makefile makes the dependencies visible.

# 5 Type $PLAYER and Related Classes

## 5.1 $PLAYER

Similar to the situation between the abstract type $CHESS_DISPLAY and the classes ASCII_DISPLAY and X_DISPLAY, the players are organized with subtyping and include as well. The abstract type $PLAYER specifies the common interface.

```
type $PLAYER is
  getmove(b:BOARD):MOVE;
  ask_pawn_xchg:CHAR;
end;
```

## 5.2 Class PLAYER

This is a class of type $PLAYER, which will *not* be used to instantiate. There will be no objects of type PLAYER. The main purpose of this class is to declare attributes and routines that are common to other classes of type $PLAYER, which **include** the implementation of this class.

The routine getmove does not provide a basic implementation. However, for consistency with the interface required by $PLAYER, a dummy implementation must be given. The routine ask_pawn_xchg provides a default implementation.

```
class PLAYER < $PLAYER is
  attr iswhite:BOOL;
  create(iswhite:BOOL):SAME is
    ret : SAME := new;
    ret.iswhite := iswhite;
    return ret;
  end;
  getmove(b:BOARD):MOVE is
    raise "PLAYER:invalid call to getmove\n";
  end;
  ask_pawn_xchg:CHAR is
    return 'Q';
  end;
end; -- of class PLAYER
```

This is a good place to look at the list of available class elements. We have already encountered routine definitions and **include** statements. Iter definitions are similar to routine definitions. All class elements can be declared **private**. Private elements can only be accessed from within the implementation of the class. Per default, class elements are public. It is worthwhile to take a closer look at the other class elements:

**const** Constant attributes are accessible by all objects in a class and may not be assigned to. Constant attributes are initialized. They are accessible even if no object of the class is created.

**shared** Shared attributes are variables that are directly accessible to all objects of a given type. They are accessible even if no object of the class is created. When only a single shared attribute is defined by a clause, it may be provided with an initializing expression which must be a constant expression. If no initialization is given, shared variables are initialized to the default.

**attr** Attributes are connected with objects. Each object of a class has an individual set of attribute variables which reflect the state of the object. Attributes are only accessible when an object has been created.

## 5.3 Class HUMAN_PLAYER

A human player will enter his move via the interface. This is coded in the routine getmove that replaces the inherited dummy implementation.

If a human player has the chance to exchange one of his pawns with a queen or a knight, the human player will enter his decision via the interface in routine ask_pawn_xchg.

```
class HUMAN < $PLAYER is
 include PLAYER;
 getmove(b:BOARD):MOVE is
  return MAIN::display.getmove(iswhite);
 end;
 ask_pawn_xchg:CHAR is
  MAIN::display.update(MAIN::board.str);
  return MAIN::display.ask_pawn_xchg;
 end;
end; -- of class HUMAN
```

## 5.4 Class MINMAX

The automatic player is represented by the class MINMAX. The class is called MINMAX, since the strategy for determining a move is based on a minmax search.

We define a couple of constants first. The boolean constants max and min are later on used to determine whether the minmax search is at a max- or at a min-level. The constant max_depth gives the maximal depth of the search tree. If max_depth is 3, then (1) all potential next moves, (2) all reactions of the opponent player and (3) all potential future reactions to these are considered.

The best moves of phase (1) are gathered in a dynamically sized list of type FLIST, as defined in the library file Library/flist.sa. FLIST will store all moves that will eventually result in the same board evaluation on level (3).

The random number generator declared in line 35 is used to select an arbitrary move from the list. MS_RANDOM_GEN is a class that is defined in the Sather Libraries. You find it in the file Library/rnd.sa The random number object is created and initialized in the **create** routine in line 40.

```
29 class MINMAX < $PLAYER is
30  include PLAYER;
31  const max : BOOL := true;
32  const min : BOOL := ~max;
33  const max_depth : INT  := 3;
34  attr bestmoves : FLIST{MOVE};
35  shared rnd : MS_RANDOM_GEN;
36  create(iswhite:BOOL):SAME is
37   ret ::= new;
38   ret.iswhite := iswhite;
```

```
39    ret.bestmoves := #FLIST{MOVE};
40    rnd := #MS_RANDOM_GEN;
41    rnd.init(4711);
42    return ret;
43  end;
```

The getmove routine at first tells the viewing user that it is "thinking" (line 46). Then it uses the routine minmax, which is described below, to find the best move. There might be more than one move that is considered to be "best". The list bestmoves stores all of these. If there are no available moves, i.e., if the list of bestmoves is empty, then the player is mate – the game is over. This is checked in line 54.

Otherwise the random number generator returns a value in $[0, 1)$. This is multiplied by the size of the list of available best moves. Before multiplication, size, which is an integer value, is cast to be of type FLTD. The product is rounded to the floor and then cast into an integer value by the routine int. The result is then used to index into the list of possible best moves.

Before returning the move to the caller, it is displayed in line 61.

```
44  getmove(board:BOARD):MOVE is
45    ret : MOVE;
46    MAIN::display.thinking(board.white_to_play);
47    if board.white_to_play then
48      -- minmax returns a value, that is nor needed. However, Sather does
49      -- require to use the return value somehow.
50      dummy ::= minmax(board,max,max_depth);
51    else
52      dummy ::= minmax(board,min,max_depth);
53    end;
54    if bestmoves.size = 0 then
55      return #MOVE("quit",board.white_to_play);
56    else
57      ret := bestmoves[(bestmoves.size.fltd * rnd.get).floor.int];
58      bestmoves.clear;
59      text : STR;
60      text := ret.from.str + "-" + ret.to.str;
61      MAIN::display.showmove(text);
62      return ret;
63    end;
64  end; -- of getmove
```

The **private** routine minmax returns a floating point value, FLT. FLT is specified in the library class FLT. See file Library/flt.sa for details.

The body of minmax has a good example of nested iter calls: The first loop (lines 74–103) considers all pieces on the board of my color. The inner loop (lines 75–102) then for each of these pieces considers target positions of potential moves. (It is explained later on, what an ordinary move is. Just ignore this flag for the time being.)

The move created in line 77 is guaranteed to be correct, i.e., the piece is of the correct color and the target position is correct with respect to the basic movement rules of chess. The only condition that is not guaranteed to hold is whether the own king is exposed to be in check after the piece is moved. This is checked in apply_move_with_own_check_test. See line 79.

After a move has been applied successfully, we either consider the possible reactions recursively (line 83), or evaluate the value of the board in line 81.

The depth-first search requires backtracking. This is done in line 100 by calling board.unapply_move.

```
65 private minmax(board:BOARD,minmax:BOOL,depth:INT):FLT is
66   move : MOVE;
67   val,bv : FLT;
68   pos : POS;
69   if minmax = max then
70     val := -1000.0;
71   else
72     val := 1000.0;
73   end;
74   loop piece::=board.my_piece!;
75     loop
76       pos :=piece.move!(board,PIECE::ordinary);
77       move := #MOVE(piece,pos);
78       move.piece := piece;
79       if board.apply_move_with_own_check_test(move) then
80         if depth = 1 then
81           bv := board.board_value;
82         else
83           bv := minmax(board,~minmax,depth - 1);
84         end;
85         -- If this move really is better than previous ones,
86         -- the list of best moves found so far is erased.
87         if depth = max_depth and (   (minmax = max and bv > val)
88                                   or (minmax = min and bv < val))
89         then
90           bestmoves.clear;
91         end;
92         -- If this move is not worse than previous ones, the move
93         -- is added to the list of best moves found so far.
94         if depth = max_depth and (   (minmax = max and bv >= val)
95                                   or (minmax = min and bv <= val))
96         then
97           val := bv;
98           bestmoves := bestmoves.push(move);
99         end;
100        board.unapply_move;
101      end;
102    end;
103  end;
104  return val;
105 end; -- of minmax
106 end; -- of class MINMAX
```

The following remark will be completely understandable only after the type $PIECE and the concrete subtypes have been presented in section 9. For reasons of completeness note that line 76 is a dispatched iter call. Depending on the concrete type of the piece:$PIECE a different iter is called.

In Sather 1.0.2 dispatched iters are not implemented. The **typecase** statement can be used to implement the intended behavior:

```
typecase piece
when PAWN   then pos:=piece.move!(board,PIECE::ordinary);
when ROOK   then pos:=piece.move!(board,PIECE::ordinary);
when KNIGHT then pos:=piece.move!(board,PIECE::ordinary);
when BISHOP then pos:=piece.move!(board,PIECE::ordinary);
when KING   then pos:=piece.move!(board,PIECE::ordinary);
when QUEEN  then pos:=piece.move!(board,PIECE::ordinary);
else
end;
```

# 6 Class MOVE

A move, i.e., an object of class MOVE stores several facts. First of all there are the from and the to position which are objects of class POS. The move knows about it being a castle move. Castle moves have from and to positions that refer to the movement of the king.

During the process of analyzing a move, further information is gathered and stored in the move object. This information is necessary to later on un-do a move. The attribute piece stores a pointer to the piece that is moved by a move. If the move kills an opponent piece, that piece can be reached by the attribute kills. The fact whether the kings have moved belongs to the status of the board. A move of a king might change that status. To preserve the fact that a particular move has changed that status, the king_chg flag has been introduced. Another flag for un-doing moves is pawn_chg. If a pawn reaches the base line of the opponent, the pawn can be exchanged to a knight or a queen. The pawn_chg flag indicates such an exchange. Although a board knows about the last move, the previous move is kept in the move object.

```
1  class MOVE is
2    attr from, to : POS;
3    attr isk_castle : BOOL;
4    attr isq_castle : BOOL;
5    attr isquit : BOOL;
6    attr piece : $PIECE;
7    attr kills : $PIECE;
8    attr king_chg : BOOL;
9    attr pawn_chg : BOOL;
10   attr prev_move : MOVE;
```

The MOVE class offers two create routines and is thus a good example of overloading. The first version of the create routine, accepts a move in standard chess notation, e.g. "a2-a3". For this version of **create** it does not matter, whether the board actually has a piece on the from position since this is checked later on. In contrast to the first version of the **create** routine, the second version deals with an existing $PIECE object. Since a piece has an actual position, only the destination position is required as parameter.

This code of the **create** routine is written rather fail safe. The given string is checked for conforming syntax. If there is an error, the from and to position of the move object remain void.

The first branch of the **if–elsif** cascade handles the q-castle (lines 21–28). The second branch handles the k-castle (lines 29–36) Then the "quit" case is considered. The fourth case (lines 39–47) and fifth case (lines 48–57) both deal with ordinary moves: They check for syntax "<p1>-<p2>" and test whether p1 and p2 refer to existing positions of the board. The string class offers a substring routine which has two parameters. It is used for example in line 40. The first argument refers to the starting position of the substring, the second argument specifies the number of characters to be returned. The difference between the fourth and the fifth case is that in the latter the the separating "-" can be omitted so that "<p1><p2>" is accepted.

```
11   create(move:STR, white_to_move:BOOL):SAME is
12     ret ::= new;
13     ret.isk_castle := false;
14     ret.isq_castle := false;
15     ret.isquit := false;
16     ret.piece := void;
17     ret.kills := void;
```

```
  ret.king_chg := false;
  ret.pawn_chg := false;
  if void(move) then return ret; end;
  if move.size >= 5 and move.head(5) = "o-o-o" then
    ret.from := #POS; ret.to := #POS;
    ret.isq_castle := true;
    if white_to_move then
      ret.from.pos := "e1"; ret.to.pos := "c1";
    else
      ret.from.pos := "e8"; ret.to.pos := "c8";
    end;
  elsif move.size >= 3 and move.head(3) = "o-o" then
    ret.from := #POS; ret.to := #POS;
    ret.isk_castle := true;
    if white_to_move then
      ret.from.pos := "e1"; ret.to.pos := "g1";
    else
      ret.from.pos := "e8"; ret.to.pos := "g8";
    end;
  elsif move.size >= 4 and move.head(4) = "quit" then
    ret.isquit     := true;
  elsif move.size >= 5 then
    str_from ::= move.substring(0,2);
    if POS::check_pos(str_from) then
      ret.from := #POS; ret.from.pos := str_from;
    end;
    str_to ::= move.substring(3,2);
    if POS::check_pos(str_to) then
      ret.to := #POS; ret.to.pos := str_to;
    end;
  elsif move.size >=4 then
    str_from ::= move.substring(0,2);
    if POS::check_pos(str_from) then
      ret.from := #POS; ret.from.pos := str_from;
    end;
    str_to   ::= move.substring(2,2);
    if POS::check_pos(str_to) then
      ret.to := #POS; ret.to.pos := str_to;
    end;
  end;
  return ret;
end; -- of first version of create
```

The routine **create** is overloaded in class MOVE, i.e., there are two routines called **create** that are distinguished by their list of formal parameters and/or return parameter. Whereas the **create** routine given above expects a string and a boolean value as parameters, the second **create** routine expects a piece and a (target) position.

```
create(piece:$PIECE, to:POS):SAME is
  ret ::= new;
  ret.isk_castle := false;
  ret.isq_castle := false;
  ret.isquit     := false;
  ret.from       := #POS;
  ret.from.pos   := piece.position.str;
  ret.to         := #POS;
  ret.to.pos     := to.str;
  ret.piece      := void;
  ret.kills      := void;
  ret.king_chg   := false;
  ret.pawn_chg   := false;
  if piece.isking then
    if piece.iswhite then
      if piece.position = "e1" and to = "c1" then
        ret.isq_castle := true;
      end;
      if piece.position = "e1" and to = "g1" then
        ret.isk_castle := true;
      end;
    else
      if piece.position = "e8" and to = "c8" then
        ret.isq_castle := true;
      end;
      if piece.position = "e8" and to = "g8" then
        ret.isk_castle := true;
      end;
    end;
  end;
  return ret;
end; -- of second version of create
isok:BOOL is
  return ~void(from) and ~void(to);
end;
end; -- of class MOVE
```

# 7 Class POS

The main secret of class POS is the internal addressing scheme for a chess board. From outside, board positions are addressed in standard chess notation, e.g., the position in the lower left corner is called "a1". Internally, POS numbers the positions row-wise from 0 to 63 which eases addressing computations. The correspondence is shown in the following tables:
External addressing scheme:

| column | 'a' | 'b' | 'c' | 'd' | 'e' | 'f' | 'g' | 'h' | row |
|---|---|---|---|---|---|---|---|---|---|
| | a8 | b8 | c8 | d8 | e8 | f8 | g8 | h8 | '8' |
| | a7 | b7 | c7 | d7 | e7 | f7 | g7 | h7 | '7' |
| | a6 | b6 | c6 | d6 | e6 | f6 | g6 | h6 | '6' |
| | a5 | b5 | c5 | d5 | e5 | f5 | g5 | h5 | '5' |
| | a4 | b4 | c4 | d4 | e4 | f4 | g4 | h4 | '4' |
| | a3 | b3 | c3 | d3 | e3 | f3 | g3 | h3 | '3' |
| | a2 | b2 | c2 | d2 | e2 | f2 | g2 | h2 | '2' |
| | a1 | b1 | c1 | d1 | e1 | f1 | g1 | h1 | '1' |

Internal addressing scheme:

| column | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | row |
|---|---|---|---|---|---|---|---|---|---|
| | 56 | 57 | 58 | 59 | 60 | 61 | 62 | 63 | 7 |
| | 48 | 49 | 50 | 51 | 52 | 53 | 54 | 55 | 6 |
| | 40 | 41 | 42 | 43 | 44 | 45 | 46 | 47 | 5 |
| | 32 | 33 | 34 | 35 | 36 | 37 | 38 | 39 | 4 |
| | 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31 | 3 |
| | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 2 |
| | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 1 |
| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 0 |

POS is capable of returning all board positions which are reachable from an POS object's position by moves in various directions. The way! iter is used for this purpose. Possible directions are vertical, horizontal, diagonal, knight jumps and so on.

POS is declared to be a subtype of $IS_EQ{POS}. The $IS_EQ type is specified in the library file Library/abstract.sa. The essential meaning of this subtype declaration is that POS is required to offer a routine with the signature is_eq(x:SAME):BOOL. The existence of this routine is checked during compilation. The analogous situation holds for $STR. This abstract type requires the existence of a routine str:STR that prints out a reasonable string representation of the object.

In lines 2–4 is an example of a rather weird form of constant declaration. All together 12 integer constants are declared. The first one (knight) is assigned the value 1, the next one (diag_up_right) is set to 2 and so on. This form of constant declaration only works for integers and requires that there is no type identifier INT. Both

    knight:INT:=1, ...

and

    knight := 'a', ...

result in errors at compile time.

The internal address of a position is stored in the private attribute absolute declared in line 8.

```
1 class POS < $IS_EQ{POS}, $STR is
2   const knight := 1, diag_up_right, diag_up_left, diag_dn_right, diag_dn_left,
3           horizontal_right, horizontal_left, vertical_up, vertical_dn,
```

```
4            north_two, south_two, ring;
5  -- The correct funtionality relies on the fact that diag_up_right to
6  -- vertical_dn are in that order. The implementation of $PIECE::move! may
7  -- depend on it.
8  private attr absolute : INT;
9  create:SAME is
10   return new;
11 end;
```

The following routines are used to handle “internal” addresses of board positions.

```
12 private pos(position:INT) is -- write routine
13   absolute := position;
14 end;
15 pos:INT is -- reader routine
16   return absolute;
17 end;
18 private row(p:POS):INT is
19   return (p.pos/8);
20 end;
21 private column(p:POS):INT is
22   return (p.pos%8);
23 end;
```

The following routines represent the “external” addressing scheme.

We discuss the routine check_pos first. The routine digit_value, which is implemented in the CHAR library class (see file Library/char.sa for details) returns the value of a character. For example '7'.digit_value=7. Note, that '\0'.int=0 and '7'.int /= 7.

The routine pos in line 39 is a good example for overloading. For dealing with the internal addressing scheme, there is already a routine called pos in line 12. That routine takes an INT as its parameter. In contrast: the following routine, accepts a STR parameter. The compiler determines, depending on the arguments which are present at a call, which of these routines has to be called.

Because of this mechanism, there cannot be two routines that have the same parameters and are different in their return types. If such a pair would be allowed, the compiler could not figure out for example which type an attribute with an implicit type declaration (e.g. A::routine) is meant to have.

Routine pos is the first occurrence of a **pre** condition in this tutorial, see line 40. The **pre** condition is a boolean expression that is checked on each call of the routine. If it is evaluated to true, the routine gets executed. Otherwise, it is a fatal error. Analogously, a **post** condition could have been declared. Note, that **pre** conditions are *not* checked by default. Checking can be invoked with the compiler flag `-pre <classes>` A frequent source of syntx error is that there may not be a semicolon behind a pre-condition because it is part of the header.

```
24 check_pos(position:STR):BOOL is
25   str : STR := position.lower;
26   if str.size /= 2 then
27     return false;
28   end;
29   row : INT := str.char(1).digit_value - 1;
30   if row < 0 or row > 7 then
```

```
31     return false;
32   end;
33   col : CHAR := str.char(0);
34   if col < 'a' or col > 'h' then
35     return false;
36   end;
37   return true;
38 end; -- of check_pos
39 pos(position:STR) -- overloaded writer routine
40 pre check_pos(position)
41 is
42   str : STR := position.lower;
43   row : INT := str.char(1).digit_value - 1;
44   col : CHAR := str.char(0);
45   case col
46     when 'a' then absolute := 0;
47     when 'b' then absolute := 1;
48     when 'c' then absolute := 2;
49     when 'd' then absolute := 3;
50     when 'e' then absolute := 4;
51     when 'f' then absolute := 5;
52     when 'g' then absolute := 6;
53     when 'h' then absolute := 7;
54   end;
55   absolute := absolute + 8 * row;
56 end; -- of pos(STR)
57 str:STR is
58   ret ::= #STR;
59   ret := ret + column;
60   ret := ret + row;
61   return ret;
62 end;
```

The routine row in line 63 is overloaded as well. The compiler can differentiate between row(INT):INT of line 18 and row:CHAR because of the different number of parameters.

In the statement in line 64 the result of the computation is an integer value. The library class INT offers two ways to convert integers into characters. The difference is best shown by means of an example. Consider the integer value 0. The conversion done by digit_char returns the character '0'. The other conversion is done by a routine called char which has the result that 0.char = '\0'.

The routine str(POS) is used internally to map an internal address, which might be different from **self**, to standard chess notation.

```
63 row:CHAR is
64   return ((absolute/8) + 1).digit_char;
65 end;
66 column:CHAR is
67   col ::= absolute%8;
68   case col
69     when 0 then return 'a';
70     when 1 then return 'b';
```

```
    when 2 then return 'c';
    when 3 then return 'd';
    when 4 then return 'e';
    when 5 then return 'f';
    when 6 then return 'g';
    when 7 then return 'h';
  end;
end;
private str(pos:INT):STR is
  ret ::= #STR;
  col ::= pos % 8;
  row ::= (pos / 8) + 1;
  case col
    when 0 then ret := "a";
    when 1 then ret := "b";
    when 2 then ret := "c";
    when 3 then ret := "d";
    when 4 then ret := "e";
    when 5 then ret := "f";
    when 6 then ret := "g";
    when 7 then ret := "h";
  end;
  ret := ret + row;
  return ret;
end; -- of str(INT)
```

The following routines return neighboring addresses in standard chess notation. If there is no existing neighboring position for a given direction, the current address is returned.

```
east:STR is
  ret ::= absolute + 1;
  if ret/8 /= absolute/8 then ret := absolute; end;
  return str(ret);
end;
west:STR is
  ret ::= absolute - 1;
  if ret/8 /= absolute/8 then ret := absolute; end;
  return str(ret);
end;
north:STR is
  ret ::= absolute + 8;
  if ret > 63 then ret := absolute; end;
  return str(ret);
end;
south:STR is
  ret ::= absolute - 8;
  if ret < 0 then ret := absolute; end;
  return str(ret);
end;
```

In addition to routines that return the address of neighboring positions in horizontal and vertical directions, there are four routines for neighboring positions on the diagonal axes.

```
northeast:STR is
  err : BOOL := false;
  ret ::= absolute + 8;
  if ret > 63 then ret := absolute; err := true; end;
  if ~err then
    ret := absolute + 1;
    if ret/8 /= absolute/8 then ret := absolute; err := true; end;
  end;
  if ~err then
    ret := absolute + 9;
  end;
  return str(ret);
end;
northwest:STR is
  err : BOOL := false;
  ret ::= absolute + 8;
  if ret > 63 then ret := absolute; err := true; end;
  if  ~err then
    ret := absolute - 1;
    if ret/8 /= absolute/8 then ret := absolute; err := true; end;
  end;
  if ~err then
    ret := absolute + 7;
  end;
  return str(ret);
end;
southeast:STR is
  err : BOOL := false;
  ret ::= absolute - 8;
  if ret < 0 then ret := absolute; err := true; end;
  if ~err then
    ret := absolute + 1;
    if ret/8 /= absolute/8 then ret := absolute; err := true; end;
  end;
  if ~err then
    ret := absolute - 7;
  end;
  return str(ret);
end;
southwest:STR is
  err : BOOL := false;
  ret ::= absolute - 8;
  if ret < 0 then ret := absolute; err := true; end;
  if ~err then
    ret := absolute - 1;
    if ret/8 /= absolute/8 then ret := absolute; err := true; end;
  end;
  if ~err then
```

```
164     ret := absolute - 9;
165   end;
166   return str(ret);
167 end;
```

Here are some equality tests. The first one is required because POS has been declared to be a subtype of $IS_EQ{POS}. The Sather compiler considers a boolean expression p=q to be syntactic sugar for the routine call p.is_eq(q). Analogously, p/=q is taken to be p.is_neq(q). If these expressions are found somewhere in the code, the corresponding routine has to be provided.

```
168 is_eq(p:SAME):BOOL is
169   return (absolute = p.pos);
170 end;
171 is_neq(p:STR):BOOL is
172   return ~is_eq(p);
173 end;
174 is_eq(p:STR):BOOL is
175   tmp ::= #POS;
176   tmp.pos := p;
177   return is_eq(tmp);
178 end;
```

The iter way! returns all reachable positions on an otherwise empty board in the specified direction.

Since this is the first occurrence of an iter declaration, some explanations are appropriate. Iters are declared similar to routines. The difference is that their name has to end with an exclamation point "!". Iters may only be called from within loop statements.

For each textual iter call, en execution state is maintained. When a loop is entered, the execution state of all iter calls is initialized. When an iter is called for the first time, the expressions for **self** and for each argument are evaluated[3].

When the iter is called, it executes the statements in its body in order. If it executes a yield statement, control and a value are returned to the caller. Subsequent calls to the iter resume execution with the statement following the **yield** statement. If an iter executes a **quit** statement or reaches the end of its body, control passes immediately to the end of the innermost enclosing loop statement in the caller and no value is returned from the iter.

The code in lines 180–183 is evaluated only at the time of the first invocation. If there are two different textual calls of way!, each one has a separate state and each will execute these code lines at the first invocation.

In line 182 the starting position of the stepping is initialized. Note that this assignment is actually a call of the private routine pos(INT). The compiler considers this expression to be equivalent to stepped.pos(absolute).

The loop in lines 184–348 is the main part of the iter. From inside the loop potential positions are returned to the caller. If no more positions are available, then a "quit" ends this loop, ends the iter and ends the loop surrounding the call to the iter.

Since most branches of the **case** statement are similar only the first case (lines 186–198) is explained in some detail. Later we will point out the differences of the branches for knight, pawn, and king moves. From the current position which is kept in stepped, the northeast neighbor is

[3] An exception are arguments which have a trailing exclamation mark themselves. These are evaluated for every call of the iter. But since this kind of argument is not used in Sather Tutorial Chess, the reader is referred to the Sather Manual [10] for further discussion.

checked. If this position is still on the board it is returned to the caller. This is done in line 192 by the **yield** statement.

After the caller has processed the new position, the next call to the iter will resume after line 192. The status is still available, i.e., stepped keeps the position, which has been returned previously. Since the only statement of the loop is this **case**, the iter will next re-execute the case and automatically re-enter this branch. (Note, the direction is not re-evaluated and remains unchanged.)

If the end of the board has been reached by moving into the northeast direction, the iter cannot return further valid position. Hence, the iter quits in the **else** branch (line 194 or 196). It does not return any position, and immediately terminates the loop in the caller.

---

```
179 way!(direction:INT):POS is
180   ret, stepped : POS;
181   stepped := #POS;
182   stepped.pos := absolute; -- starting position
183   count : INT := 0;
184   loop
185     case direction
186     when diag_up_right then
187       if stepped.column < 'h' then
188         if stepped.row < '8' then
189           stepped.pos := stepped.northeast;
190           ret := #POS;
191           ret.pos := stepped.pos;
192           yield ret;
193         else
194           quit;
195         end;
196       else
197         quit;
198       end;
199     when diag_up_left then
200       if stepped.column > 'a' then
201         if stepped.row > '1' then
202           stepped.pos := stepped.southwest;
203           ret := #POS;
204           ret.pos := stepped.pos;
205           yield ret;
206         else
207           quit;
208         end;
209       else
210         quit;
211       end;
212     when diag_dn_right then
213       if stepped.column < 'h' then
214         if stepped.row > '1' then
215           stepped.pos := stepped.southeast;
216           ret := #POS;
217           ret.pos := stepped.pos;
218           yield ret;
219         else
```

```
      quit;
    end;
  else
    quit;
  end;
when diag_dn_left then
  if stepped.column > 'a' then
    if stepped.row < '8' then
      stepped.pos := stepped.northwest;
      ret := #POS;
      ret.pos := stepped.pos;
      yield ret;
    else
      quit;
    end;
  else
    quit;
  end;
when vertical_up then
  if stepped.row < '8' then
    stepped.pos := stepped.north;
    ret := #POS;
    ret.pos := stepped.pos;
    yield ret;
  else
    quit;
  end;
when vertical_dn then
  if stepped.row > '1' then
    stepped.pos := stepped.south;
    ret := #POS;
    ret.pos := stepped.pos;
    yield ret;
  else
    quit;
  end;
when horizontal_right then
  if stepped.column < 'h' then
    stepped.pos := stepped.east;
    ret := #POS;
    ret.pos := stepped.pos;
    yield ret;
  else
    quit;
  end;
when horizontal_left then
  if stepped.column > 'a' then
    stepped.pos := stepped.west;
    ret := #POS;
    ret.pos := stepped.pos;
    yield ret;
```

```
271   else
272     quit;
273   end; -- way! will be continued ...
```

The branch of the **case** statement that computes the new position of a knight in lines 274–296 is somewhat different. Instead of using a current position (called stepped), the new positions are always computed relative to the starting position.

A white pawn (case north_two, lines 297–307) may move one or to steps to the north depending on the staring row. A black pawn (case south_two, lines 308–318) may move one or to steps to the south depending on the staring row. A king (case ring, lines 319–342) can reach all 8 positions on the ring around his staring position.

```
274 when knight then
275   ret := #POS;
276   case count
277     when 0 then ret.pos := absolute + 6;
278     when 1 then ret.pos := absolute - 6;
279     when 2 then ret.pos := absolute + 10;
280     when 3 then ret.pos := absolute - 10;
281     when 4 then ret.pos := absolute + 15;
282     when 5 then ret.pos := absolute - 15;
283     when 6 then ret.pos := absolute + 17;
284     when 7 then ret.pos := absolute - 17;
285   else
286     quit;
287   end;
288   count := count + 1;
289   if ret.pos <= 63 and ret.pos >= 0
290      and column(ret) <= column(self) + 2
291      and column(self) - 2 <= column(ret)
292      and row(ret)    <= row(self) + 2
293      and row(self) - 2    <= row(ret)
294   then
295     yield ret;
296   end;
297 when north_two then
298   if count < 2 and stepped /= stepped.north then
299     stepped.pos := stepped.north;
300     ret := #POS;
301     ret.pos := stepped.pos;
302     count := count + 1;
303     yield ret;
304     if row /= '2' then quit; end;
305   else
306     quit;
307   end;
308 when south_two then
309   if count < 2 and stepped /= stepped.south then
310     stepped.pos := stepped.south;
311     ret := #POS;
```

```
        ret.pos := stepped.pos;
        count := count + 1;
        yield ret;
        if row /= '7' then quit; end;
      else
        quit;
      end;
    when ring then
      ret := #POS;
      case count
        when 0 then ret.pos := north;
        when 1 then ret.pos := south;
        when 2 then ret.pos := east;
        when 3 then ret.pos := west;
        when 4 then ret.pos := northeast;
        when 5 then ret.pos := northwest;
        when 6 then ret.pos := southeast;
        when 7 then ret.pos := southwest;
      else
        quit;
      end;
      count := count + 1;
      if   ret.pos <= 63 and ret.pos >= 0
        and ret.pos /= absolute
         and column(ret) <= column(self) + 1
         and column(self) - 1 <= column(ret)
         and row(ret)    <= row(self) + 1
         and row(self) - 1    <= row(ret)
       then
        yield ret;
      end;
    else
      -- The else case was put in for reasons of
      -- fail safe program development.
      raise "POS:way! invalid case\n";
    end; -- of case
  end; -- of loop
 end; -- of way!
end; -- of class POS
```

# 8 Class BOARD

The two array whitexpieces and blackpieces store the pieces in the game. A piece is an object of type $PIECE which is explained below. Since both arrays are private, it is a secret of the board implementation in which way pieces are stored.

The board stores information about which color is to play (white_to_play) and about the last move (last_move). Moreover, the board knows whether the white or black king has been moved. This information is necessary, because castle moves are only allowed if the king has not been moved before.

```
class BOARD is
 private attr whitepieces : ARRAY{$PIECE};
 private attr blackpieces : ARRAY{$PIECE};
         attr white_to_play : BOOL;
         attr last_move : MOVE;
         attr white_K_moved : BOOL;
         attr black_K_moved : BOOL;
 create:SAME is
  ret::=new;
  ret.set_up;
  return ret;
 end;
```

The private routine set_up initializes the board. 16 white and 16 black pieces are placed onto the board, the first player is set to be white, both kings have not moved.

```
 private set_up is
  position ::= #POS;
  white_to_play := true;
  -- set up white pieces
  whitepieces := #(16);
  position.pos := "a2";
  loop i::=0.upto!(7);
   whitepieces[i] := #PAWN(position,PIECE::white);
   position.pos := position.east;
  end;
  position.pos := "a1"; whitepieces[8] := #ROOK(position,PIECE::white);
  position.pos := "b1"; whitepieces[9] := #KNIGHT(position,PIECE::white);
  position.pos := "c1"; whitepieces[10] := #BISHOP(position,PIECE::white);
  position.pos := "d1"; whitepieces[11] := #QUEEN(position,PIECE::white);
  position.pos := "e1"; whitepieces[12] := #KING(position,PIECE::white);
  position.pos := "f1"; whitepieces[13] := #BISHOP(position,PIECE::white);
  position.pos := "g1"; whitepieces[14] := #KNIGHT(position,PIECE::white);
  position.pos := "h1"; whitepieces[15] := #ROOK(position,PIECE::white);
  -- set up black pieces
  blackpieces := #(16);
  position.pos := "a7";
  loop i::=0.upto!(7);
   blackpieces[i] := #PAWN(position,PIECE::black);
   position.pos := position.east;
```

```
37   end;
38   position.pos := "a8"; blackpieces[8] := #ROOK(position,PIECE::black);
39   position.pos := "b8"; blackpieces[9] := #KNIGHT(position,PIECE::black);
40   position.pos := "c8"; blackpieces[10] := #BISHOP(position,PIECE::black);
41   position.pos := "d8"; blackpieces[11] := #QUEEN(position,PIECE::black);
42   position.pos := "e8"; blackpieces[12] := #KING(position,PIECE::black);
43   position.pos := "f8"; blackpieces[13] := #BISHOP(position,PIECE::black);
44   position.pos := "g8"; blackpieces[14] := #KNIGHT(position,PIECE::black);
45   position.pos := "h8"; blackpieces[15] := #ROOK(position,PIECE::black);
46   white_K_moved := false;
47   black_K_moved := false;
48   last_move := void;
49   MAIN::display.redraw(self.str);
50 end;
```

Several iters are needed to return all pieces on the board that fulfill a certain condition.

The first iter whitepiece! returns all white pieces, which are still alive. For this purpose, it makes use of the iter elt! in line 52. The iter is provided by the ARRAY library class (see file Library/array.sa). If elt! yields an element, this element is yield to the caller if it fulfills the conditions. However, if elt! quits, this loop is terminated as well, no element is returned to the caller.

```
51 private whitepiece!:$PIECE is
52   loop p ::= whitepieces.elt!;
53     if ~void(p) and p.alive then yield p; end;
54   end;
55 end;
56 private blackpiece!:$PIECE is
57   loop
58     p ::= blackpieces.elt!;
59     if ~void(p) and p.alive then yield p; end;
60   end;
61 end;
```

The nesting depth of iters can be increased even further, as shown in my_piece below: Within whitepiece! the iter elt! is used. An element found by elt! is returned via whitepiece! and then returned to the caller of my_piece!. Similarly, a quit of elt!, induces a quit of whitepiece!, which in turn results in a quit of my_piece!. The latter terminates the loop, that must surround every call of my_piece! in the caller.

```
62 my_piece!:$PIECE is
63   if white_to_play then
64     loop
65       yield whitepiece!;
66     end;
67   else
68     loop
69       yield blackpiece!;
70     end;
71   end;
72 end;
```

```
73 private opp_piece!:$PIECE is
74   if white_to_play then
75     loop
76       yield blackpiece!;
77     end;
78   else
79     loop
80       yield whitepiece!;
81     end;
82   end;
83 end;
84 piece!:$PIECE is
85   loop
86     yield whitepiece!;
87   end;
88   loop
89     yield blackpiece!;
90   end;
91 end;
```

One of the secrets of the BOARD implementation is the way pieces are stored. For internal purposes it is necessary, to find out at which position of the arrays a particular piece is stored.

In the **private** routine index we use a **post** condition. To assure that the piece p is (dead or alive) on board we test whether the return value **result** is set appropriately, i.e., whether **result** is between 0 and 15 upon return. Note, that there may not be a semicolon behind a **post** condition. The conditions get checked before the routine returns. To access the value that will be returned, Sather provides the predefined **results** expression. The type of **results** is determined by the result type of the routine. If checking is desired, it has to be activated with the compiler flag -post <classes>.

The loop (line 97–104) is an excellent example of a loop that is controlled by multiple iters. The first two iters are defined in the ARRAY library class. The iter ind! (line 98) returns the existing indexes of array. As explained above, elt! (line 99) returns the corresponding array elements. For each iteration of the loop the following condition holds: whitepieces[i] = q. Both iters can be expected to yield the same number of times. If the end of the array is encountered, the call to ind! will terminate the loop; elt! will not be called.

However, if the desired piece is found, it is not necessary, to continue the search. To terminate the loop immediately, the predefined iter **break!** is called in line 102, which will always execute a **quit** statement.

The same search is implemented differently in the **else** branch (line 106). Here we use the library routine index_of provided in the ARRAY class. (See file Library/array.sa for details.)

```
92  private index(p:$PIECE):INT
93  post result.is_bet(0,15)
94  is
95    ret : INT := -1;
96    if p.iswhite then
97      loop
98        i::= whitepieces.ind!;
99        q::= whitepieces.elt!;
100       if p.position = q.position then
```

```
      ret := i;
      break!;
    end;
  end; -- of loop
else
  ret := blackpieces.index_of(p);
end;
return ret;
end; -- private index
```

To check whether there is a piece on a given position of the board the following routines are provided:

```
has_piece(pos:POS):BOOL is
  ret : BOOL := false;
  loop p::=piece!;
    if p.position = pos then ret := true; end;
  end;
  return ret;
end;
has_white_piece(pos:POS):BOOL is
  ret : BOOL := false;
  loop p::=whitepiece!;
    if p.position = pos then ret := true; end;
  end;
  return ret;
end;
has_black_piece(pos:POS):BOOL is
  ret : BOOL := false;
  loop p::=blackpiece!;
    if p.position = pos then ret := true; end;
  end;
  return ret;
end;
has_my_piece(pos:POS):BOOL is
  if white_to_play then
    return has_white_piece(pos);
  else
    return has_black_piece(pos);
  end;
end;
```

The following two routines return a pointer to a piece at a given position of the board. The routine comes in two versions. The latter can process POS arguments by reducing them to STR parameters which are then processed by the first version.

```
piece(str:STR):$PIECE is
  ret : $PIECE;
  position ::= #POS;
  position.pos := str;
  loop p::=piece!;
```

```
    if p.position = position then ret := p; end;
  end;
  return ret;
end;
piece(p:POS):$PIECE is
  return piece(p.str);
end;
```

For interface purposes, a board can represent the status of all pieces in an ASCII representation. The character array is used to transmit the board situation to the ASCII_DISPLAY and via the X_DISPLAY to the external class XCW.

```
str:ARRAY{CHAR} is
  ret::=#ARRAY{CHAR}(65);
  loop
    ret[0.upto!(63)] := ' ';
  end;
  ret[64] := '\0';
  loop p::=self.whitepiece!;
    if ~void(p) and p.alive then
      ret[p.position.pos] := p.fig;
    end;
  end;
  loop p::=self.blackpiece!;
    if ~void(p) and p.alive then
      ret[p.position.pos] := p.fig.lower;
    end;
  end;
  return ret;
end;
```

After these helper routines and iters have been implemented, the central routines are presented. The routine pos_in_check tests whether a given position could be reached in the next move by the opponent.

In this routine there is again a good example of nested iter calls: The first loop (line 172–179) considers all pieces of the opponent player. The inner loop (line 173–178) then for each of these pieces considers target positions of potential moves. (Is is explained later on, what a move is if the flag for_check_test is set. Just ignore the flag for the time being.)

The call piece.move!() in line 174is a dispatched iter. See page 24 for an alternative implementation that works with earlier releases of the Sather 1.0 compiler.

```
168 pos_in_check(p:POS):BOOL is
169   ret : BOOL;
170   pos : POS;
171   ret := false;
172   loop piece::=opp_piece!;
173     loop
174       pos :=piece.move!(self, PIECE::for_check_test);
175       if p=pos then
176         ret := true;
```

```
      break!;
    end;
  end;
  if ret then break!; end;
 end;
 return ret;
end; -- of pos_in_check
```

The routine my_king_isin_check returns true if the king of the current color (white_to_play) is in check. After an otherwise valid move of a piece, the own king is not allowed to be exposed and to be in check.

```
my_king_isin_check:BOOL is
 piece : $PIECE;
 loop
  piece := my_piece!;
  until!(piece.isking);
 end;
 return pos_in_check(piece.position);
end; -- of my_king_isin_check
```

Boolean expressions are evaluated with a short-circuit semantics. For an and this means that the second operand is only evaluated if the first operand was true. For an or the second operand is evaluated only if the first one was false. In routine check_n_apply_move we make use of this to ensure that a move is applied to a board only if it is valid.

Routine move_valid_so_far checks whether a given move is valid with respect to the current state of the board. The only circumstance which is not checked is whether the move would expose the own king to be in check.

```
check_n_apply_move(move:MOVE):BOOL is
 return (move_valid_so_far(move) and apply_move_with_own_check_test(move));
end; -- of check_n_apply_move
private move_valid_so_far(move:MOVE):BOOL
pre ~move.isquit
is
 ret : BOOL := false;
 -- A valid move must start at a position where one of my pieces is....
 if has_my_piece(move.from) then
  p::=piece(move.from);
  -- ... and it must be a valid move with respect to the mobility of the
  -- piece at the current state of the board.
  if p.valid_move(move.to,self) then
   ret := true;
   -- Since the move seems to be valid, the moving piece is stored
   -- in the move object. That eases future access to the moving piece
   -- and allows for un-doing of moves.
   move.piece := p;
  end;
 end;
 return ret;
```

```
213 end; -- of move_valif
```

The move is applied to the board in routine apply_move_with_own_check_test. The routine returns false, and leaves the state of the board unchanged, if an otherwise valid move would expose the own king to be in check.

First of all in lines 221-241 it is checked whether the move would kill an opponent piece. The normal circumstances for this are that the moving piece moves to a position that is occupied by an opponent piece. Chess has one special rule due to which a piece can be killed without moving to its former position. It is called an "en passant" move. This special case can only occur if two pawns are involved. My pawn can kill an opponent pawn that sits immediately east or west of my pawn, if the other pawn has done an initial double move in the immediately preceding move. (That's why the last move is considered to be part of the state of a board.). If these conditions hold, my pawn can move diagonal so that his new position is "behind" the opponent pawn.

Special action is required in case of castle moves. A castle move works as follows. If the king and a rook both are in their initial positions, if there is no piece in between them, if the king has not been moved in the game, and if the two positions next to the king in the direction toward the rook are not in check, then the king moves two positions towards the rook and then the rook jumps over the king and is put immediately next to the king. A castle move is a k-castle, if the king moves to the rook whose initial position is closer. Otherwise it is called q-castle, because due to the initial queen position, the distance to the rook is larger. Chess only allows castle moves, if the king has not been moved earlier in the game. The board keeps track of king moves in the two flags white_K_moved and black_K_moved. To enable un-doing of moves, a move knows whether it causes a change of a K_moved flag. See lines 254–268 for the K_moved flags and lines 269–286 for the implementation of castle moves.

Another special rule in chess allows to exchange a pawn against a queen or a knight when it reaches the base line of the opponent. Theoretically, a player could have 9 queens. This rule is implemented in lines 254–268.

```
214 apply_move_with_own_check_test(move:MOVE):BOOL
215 pre ~move.isquit and move_valid_so_far(move) and ~void(move.piece)
216 is
217   ret : BOOL := true; -- Will be false if the move is invalid due to
218                       -- exposure of own "king in chess"
219   p:$PIECE:=move.piece; -- to be moved
220   r:$PIECE;             -- may be killed
221   -- Case 1: Kill with normal move
222   r := piece(move.to);  -- If it exists, it can only be opponent piece.
223                         -- Otherwise the move would not be valid.
224   -- Case 2: En Passant.
225   if     void(r) and ~void(last_move) and p.ispawn
226      and ~void(last_move.piece) and last_move.piece.ispawn
227      and (   last_move.to = p.position.east
228           or last_move.to = p.position.west)
229   then
230     if  (     p.iswhite and white_to_play
231           and p.position.row = '5' and last_move.from.row = '7')
232        or (   ~p.iswhite and ~white_to_play
233           and p.position.row = '4' and last_move.from.row = '2')
234     then
235       r := last_move.piece;
```

```
    end;
  end;
  if ~void(r) then
    move.kills := r;
    r.alive := false;
  end;
  p.update_position(move.to);
  -- Deal with king moves.
  if p.isking then
    if white_to_play and ~white_K_moved then
      white_K_moved := true;
      move.king_chg := true;
    end;
    if ~white_to_play and ~black_K_moved then
      black_K_moved := true;
      move.king_chg := true;
    end;
  end;
  -- Deal with pawn exchange.
  if   (p.ispawn and  p.iswhite and  white_to_play and p.position.row='8')
     or (p.ispawn and ~p.iswhite and ~white_to_play and p.position.row='1')
  then
    case MAIN::player.ask_pawn_xchg
    when 'Q' then
      whitepieces[index(p)] := #QUEEN(p.position,PIECE::white);
    when 'K' then
      whitepieces[index(p)] := #KNIGHT(p.position,PIECE::white);
    else
      -- Do it fails safe.
      raise "BOARD:apply_move:pawn_exchange invalid case entry\n"
    end;
    move.pawn_chg := true;
  end;
  -- Deal with castles.
  if move.isq_castle then
    if white_to_play then
      rook ::= piece("a1");
      rook.update_position("d1");
    else
      rook ::= piece("a8");
      rook.update_position("d8");
    end;
  elsif move.isk_castle then
    if white_to_play then
      rook ::= piece("h1");
      rook.update_position("f1");
    else
      rook ::= piece("h8");
      rook.update_position("f8");
    end;
  end;
```

```
  move.prev_move := last_move;
  last_move := move;
  -- Check whether my king is in check after application of the move
  if my_king_isin_check then
    -- Although otherwise correct this is an invalid move.
    -- The original state of the board is reconstructed by calling
    -- unapply_move.
    ret := false;
    unapply_move;
  end;
  white_to_play := ~white_to_play;
  return ret;
end; -- of apply_move
```

The routine unapply_move uses the information that is stored in last_move to replay the move, i.e., restore the board to the state it had before the application of that move. It depends on the fact that last_move is a valid move except that the king might be in check afterwards.

```
unapply_move is
  -- Restore killed opponent piece
  if ~void(last_move.kills) then
    last_move.kills.alive := true;
  end;
  -- Restore pawn exchange
  if last_move.pawn_chg then
    newpiece ::= piece(last_move.piece.position);
    whitepieces[index(newpiece)] := last_move.piece;
  end;
  -- Restore move
  last_move.piece.update_position(last_move.from);
  if last_move.king_chg then
    if white_to_play then
      white_K_moved := false;
    else
      black_K_moved := false;
    end;
  end;
  -- Restore castle
  if last_move.isq_castle then
    if white_to_play then
      rook ::= piece("d1");
      rook.update_position("a1");
    else
      rook ::= piece("d8");
      rook.update_position("a8");
    end;
  elsif last_move.isk_castle then
    if white_to_play then
      rook ::= piece("f1");
      rook.update_position("h1");
```

```
    else
      rook ::= piece("f8");
      rook.update_position("h8");
    end;
  end;
  last_move := last_move.prev_move;
  white_to_play := ~white_to_play;
end; -- of unapply_move
```

For the automatic player, there must be a way to assign a worth to a board. This is done as follows. Compute the sum of the worths of all white pieces on the board. Similar, compute the worth of all black pieces. The value of the board is the ratio of the two values.

The routine board_value returns a floating point value, FLT, which is specified in the FLT library. (See file Library/flt.sa for details.)

More complex evaluation functions are known and can be used to replace the simple function board_value. For example, the degree of freedom the pieces have in their movement is an interesting aspect that might be considered in the evaluation function.

```
  board_value:FLT is
    white_value : INT := 0;
    black_value : INT := 0;
    loop p::= whitepiece!;
      white_value := white_value + p.worth;
    end;
    loop p::= blackpiece!;
      black_value := black_value + p.worth;
    end;
    return white_value.flt/black_value.flt;
  end; -- of board_value
end; -- of class BOARD
```

# 9 Type $PIECE and Related Classes

For the pieces the same structure of abstract and concrete types is used that has been used before for players and displays. The abstract type $PIECE specifies the common interface. The concrete type or class PIECE is *not* used to create objects, but provides common implementations that are inherited by the real pieces (i.e., by classes PAWN, ROOK, KNIGHT, BISHOP, QUEEN, and KING).

## 9.1 Type $PIECE

```
type $PIECE is
  alive:BOOL;
  alive(set:BOOL);
  worth:INT;
  iswhite:BOOL;
  position:POS;
  valid_move(to:POS,board:BOARD):BOOL;
  update_position(position:POS);
  update_position(position:STR);
  move!(b:BOARD,to_piece:BOOL):POS;
  fig:CHAR;
  ispawn : BOOL;
  isrook : BOOL;
  isking : BOOL;
end; -- of type $PIECE
```

## 9.2 Class PIECE

```
class PIECE < $PIECE is
  -- General constants that are used throughout the descendants of $PIECE
  const white : BOOL := true;
  const black : BOOL := ~white;
  const ordinary : BOOL := false;
  const for_check_test : BOOL := true;   -- alters behavior of move!
  -- Attributes that are specific to a PIECE
  attr alive : BOOL;
  attr iswhite : BOOL;
  attr position : POS;
  -- Constants that are specific to a PIECE
  const worth : INT := 0;
  const fig : CHAR := ' ';
  const ispawn : BOOL := false;
  const isking : BOOL := false;
  const isrook : BOOL := false;
  create(pos:POS,iswhite:BOOL):SAME is
    ret ::= new;
    ret.position := #POS;
    ret.position.pos := pos.str;
    ret.iswhite := iswhite;
```

```
    ret.alive := true;
    return ret;
  end;
  private same_color(b:BOARD,p:POS):BOOL
  pre b.has_piece(p)
  is
    white_piece_on_pos :BOOL:= b.has_white_piece(p);
    if   ( iswhite and  white_piece_on_pos)
       or (~iswhite and ~white_piece_on_pos) then
      return true;
    else
      return false;
    end;
  end;
```

The following routine valid_move checks whether a given move is valid for a given board situation This is done as follows. For the from position, all valid moves are generated by calling the iter move! in line 53. It is then checked, whether the given move is in the returned set of valid moves.

```
51   valid_move(to:POS,board:BOARD):BOOL is
52     ret : BOOL := false;
53     loop valid_to::=move!(board,ordinary);
54       if to=valid_to then ret:=true; break!; end;
55     end;
56     return ret;
57   end;
58   update_position(p:POS) is
59     position.pos:=p.str;
60   end;
61   update_position(pos:STR) is
62     position.pos:=pos;
63   end;
64   move!(b:BOARD,mode:BOOL):POS is
65     raise "PIECE:dummy code (move!) called";
66   end;
67 end; -- of class PIECE
```

## 9.3 Class BISHOP

First, constants are redefined that have values which differ from those given in the PIECE implementation. The iter move! returns all valid moves given a board with other pieces. The outer loop (lines 75–86) will check the following directions: diag_up_right, diag_up_left, diag_dn_right, and diag_dn_left. In the inner loop (lines 76–85) all positions are computed a piece could reach in a direction set by the outer loop. A position returned by way! in line 76 is valid as long as there is no other piece occupying that position.

If there is another piece on the position returned by way! this cannot be a piece of the same color. However, for a check-test, the occupied position is checked by the moving piece.

```
class BISHOP < $PIECE is
 include PIECE;
 -- Constants that are different from PIECE implementation:
 const worth : INT  := 3;
 const fig : CHAR := 'B';
 move!(b:BOARD,mode:BOOL):POS is
  to : POS;
  loop direction::=POS::diag_up_right.upto!(POS::diag_dn_left);
   loop to:=position.way!(direction);
    if ~b.has_piece(to) then
     yield to;
    elsif same_color(b,to) and mode=ordinary then
     break!;
    else
     yield to;
     break!
    end;
   end;
  end;
 end; -- of move!
end; -- of class BISHOP
```

## 9.4 Class ROOK

The implementation of class ROOK is very similar to the code of BISHOP.

```
class ROOK < $PIECE is
 include PIECE;
 -- Constants that are different from PIECE implementation:
 const worth : INT  := 5;
 const fig : CHAR := 'R';
 const isrook : BOOL := true;
 move!(b:BOARD,mode:BOOL):POS is
  -- returns all valid moves given a board with other pieces
  to : POS;
  -- This loop will check the following directions:
  -- horizontal_right, horizontal_left, vertical_up, vertical_dn
  loop direction::=POS::horizontal_right.upto!(POS::vertical_dn);
   loop to:=position.way!(direction);
    if ~b.has_piece(to) then
     yield to;
    elsif same_color(b,to) and mode=ordinary then break!;
     else
     yield to;
     break!
    end;
   end;
  end;
 end; -- of move!
```

```
end; -- of class ROOK
```

## 9.5 Class QUEEN

The implementation of class QUEEN is very similar to the code of BISHOP.

```
class QUEEN < $PIECE is
 include PIECE;
 -- Constants that are different from PIECE implementation:
 const worth : INT  := 9;
 const fig : CHAR := 'Q';
 move!(b:BOARD,mode:BOOL):POS is
  -- returns all valid moves given a board with other pieces
  to : POS;
  -- This loop will check the following directions:
  -- diag_up_right, diag_up_left, diag_dn_right, diag_dn_left
  -- horizontal_right, horizontal_left, vertical_up, vertical_dn
  -- It is a combination of rook and bishop.
  loop direction::=POS::diag_up_right.upto!(POS::vertical_dn);
   loop to:=position.way!(direction);
    if ~b.has_piece(to) then
     yield to;
    elsif same_color(b,to) and mode = ordinary then break!;
    else
     yield to;
     break!
    end;
   end;
  end;
 end; -- of move!
end; -- of class QUEEN
```

## 9.6 Class KNIGHT

The body of the loop is slightly different to the one used for ROOK, BISHOP and QUEEN. Above, the inner loop terminated as soon as a position was encountered that was blocked by another piece. For KNIGHT (and later on for KING) *all* potential position have to be considered.

```
class KNIGHT < $PIECE is
 include PIECE;
 -- Constants that are different from PIECE implementation:
 const worth : INT  := 3;
 const fig : CHAR := 'N';
 move!(b:BOARD,mode:BOOL):POS is
  -- returns all valid moves given a board with other pieces
  to : POS;
  loop to:=position.way!(POS::knight);
   if b.has_piece(to) and same_color(b,to) and mode = ordinary then
```

```
148       -- skip this move
149     else
150       yield to;
151     end;
152   end;
153  end; -- of move!
154 end; -- of class KNIGHT
```

## 9.7 Class PAWN

The iter move! is different for the pawns: In ordinary mode, straight moves, diagonal moves and "en passant" moves must be considered. In check_test mode, straight moves are irrelevant. The implementation of move! is divided in two sections by an **if** statement. In the **then** branch (line 164–215) the potential moves of white pawns are computed. The **else** branch (lines 216–267) is devoted to the black pawns.

```
155 class PAWN < $PIECE is
156  include PIECE;
157  -- Constants that are different from PIECE implementation:
158  const worth : INT  := 1;
159  const fig : CHAR := 'P';
160  const ispawn : BOOL := true;
161  move!(b:BOARD,mode:BOOL):POS is
162   -- returns all valid moves given a board with other pieces
163   to : POS;
164   if iswhite then
165    if mode = ordinary then
166     -- vertical steps
167     loop to:=position.way!(POS::north_two);
168      if b.has_piece(to) then     -- position and continued way blocked
169       break!;
170      end;
171      yield to;
172     end;
173    end;
174    -- diag_up
175    if position.column < 'h' then
176     to:=#POS;
177     to.pos := position.northeast;
178     if mode = for_check_test then
179      yield to;
180     else
181      if b.has_black_piece(to) then
182       yield to;
183      end;
184     end;
185    end;
186    -- diag_dn
187    if position.column > 'a' then
188     to:=#POS;
```

```
    to.pos := position.northwest;
    if mode = for_check_test then
      yield to;
    else
      if b.has_black_piece(to) then
        yield to;
      end;
    end;
  end;
  -- en passant
  if    position.row = '5'
     and ~void(b.last_move)
     and b.last_move.from.row = '7'
     and (   b.last_move.to = position.east
          or b.last_move.to = position.west)
     and ~void(b.last_move.piece) and b.last_move.piece.ispawn
  then
    if mode = for_check_test then
      yield b.last_move.to;
    else
      to := #POS;
      to.pos := b.last_move.to.north;
      yield to;
    end;
  end;
  -- no more moves;
  quit;
else -- i.e. if isblack
  if mode = ordinary then
    -- vertical steps
    loop to:=position.way!(POS::south_two);
      if b.has_piece(to) then    -- position and continued way blocked
        break!;
      end;
      yield to;
    end;
  end;
  -- diag_up
  if position.column > 'a' then
    to:=#POS;
    to.pos := position.southwest;
    if mode = for_check_test then
      yield to;
    else
      if b.has_white_piece(to) then
        yield to;
      end;
    end;
  end;
  -- diag_dn
  if position.column< 'h' then
```

```
240       to:=#POS;
241       to.pos := position.southeast;
242       if mode = for_check_test then
243         yield to;
244       else
245         if b.has_white_piece(to) then
246           yield to;
247         end;
248       end;
249     end;
250     -- en passant
251     if    position.row = '4'
252       and ~void(b.last_move)
253       and b.last_move.from.row = '2'
254       and (   b.last_move.to = position.east
255            or b.last_move.to = position.west)
256       and ~void(b.last_move.piece) and b.last_move.piece.ispawn
257     then
258       if mode = for_check_test then
259         yield b.last_move.to;
260       else
261         to := #POS;
262         to.pos := b.last_move.to.south;
263         yield to;
264       end;
265     end;
266     quit;
267   end;
268  end; -- of move!
269 end; -- of class PAWN
```

## 9.8 Class KING

In the iter move! of the KING up to 8 neighboring positions have to be analyzed. As usual, this is done by using the way! iter provided by the POS class. Furthermore, the king might be able to do a castle move. If the preconditions of castle moves are fulfilled, the new position of the king is **yield**. Castle moves are analyzed separately for the white king in lines 290–321 and for the black king in lines 322–352.

```
270 class KING < $PIECE is
271  include PIECE;
272  -- Constants that are different from PIECE implementation:
273  const worth : INT  := 100;    -- compared to the worth of other pieces
274                                -- the king has an infinite worth
275  const fig : CHAR := 'K';
276  const isking : BOOL := true;
277  move!(b:BOARD,mode:BOOL):POS is
278   -- returns all valid moves given a board with other pieces
279   to   : POS;
280   loop to:=position.way!(POS::ring);
```

```
  if b.has_piece(to) and same_color(b,to) and mode = ordinary then
    -- skip this move
  else
    if mode = for_check_test or ~b.pos_in_check(to) then
      yield to;
    end;
  end;
end;
-- castle moves
spot1, spot2, spot3, rook : $PIECE;
if b.white_to_play and ~b.white_K_moved and position = "e1" then
  -- q-castle
  spot1:= b.piece("d1"); spot2:= b.piece("c1"); spot3:= b.piece("b1");
  rook := b.piece("a1");
  if    ~void(rook) and rook.isrook and rook.alive
     and void(spot1) and void(spot2) and void(spot3)
  then
    to := #POS;
    to.pos := "d1";
    if ~b.pos_in_check(to) then
      to.pos := "c1";
      if ~b.pos_in_check(to) then
        yield to;
      end;
    end;
  end;
  -- k-castle
  spot1:= b.piece("f1"); spot2:= b.piece("g1"); rook := b.piece("h1");
  if    ~void(rook) and rook.isrook and rook.alive
     and void(spot1) and void(spot2)
  then
    to := #POS;
    to.pos := "f1";
    if ~b.pos_in_check(to) then
      to.pos := "g1";
      if ~b.pos_in_check(to) then
        yield to;
      end;
    end;
  end;
end; -- castle moves of white king
if ~b.white_to_play and ~b.black_K_moved and position = "e8" then
  -- q-castle
  spot1:= b.piece("d8"); spot2:= b.piece("c8"); spot3:= b.piece("b8");
  rook := b.piece("a8");
  if    ~void(rook) and rook.isrook and rook.alive
     and void(spot1) and void(spot2) and void(spot3)
  then
    to := #POS;
    to.pos := "d8";
    if ~b.pos_in_check(to) then
```

```
        to.pos := "c8";
        if ~b.pos_in_check(to) then
          yield to;
        end;
      end;
    end;
    -- k-castle
    spot1:= b.piece("f8"); spot2:= b.piece("g8"); rook := b.piece("h8");
    if    ~void(rook) and rook.isrook and rook.alive
       and void(spot1) and void(spot2)
    then
      to := #POS;
      to.pos := "f8";
      if ~b.pos_in_check(to) then
        to.pos := "g8";
        if ~b.pos_in_check(to) then
          yield to;
        end;
      end;
    end;
  end; -- castle move of black king
 end; -- of move!
end; -- of class KING
```

## 10 Suggested Execises

- Extend Sather Tutorial Chess to print out all moves of the game in standard chess notation after the game is over.
- If the user decides to have a computer player, the random number generator always is initialized with the same seed. Extend Sather Tutorial Chess to ask the user for his name. Then from this name compute a seed to initialize the random number generator.
- Introduce a new subtype of $PLAYER that inherits the implementation of MINMAX. Call this class ALPHABETA and implement an Alpha-Beta-Search to improve the expertise of the automatic player. You might want to change the routine setup of MAIN to create an ALPHABETA player instead of a MINMAX player.
- Change POS to be a **value** class. Instead of having the internal addressing scheme that numbers the positions of the board from 0 to 63 in variable absolute, the positions should be represented with two integers, one for the row number and the other for the number of the column. Obviously, nearly all routines in POS have to be changed to reflect that choice. Other than that the code is relatively independent of the implementation of POS. There might be some problems when POS objects are tested to be void. Furthermore, the routine is the only place outside of board.str that knows about the internal addressing used in POS. Note, that the new internal addressing eases the complexity of the computation of neighboring elements slightly. Instead of divisions and modulo operations, a routine is_off_board could be used to deal with all the necessary plausibility testing.
- See section 1.4 for further suggestions.

# References


[1] Robert Henderson and Benjamin Zorn. A comparison of object-oriented programming in four modern languages. Technical Report CU-CS-641-93, University of Colerado, Boulder, July 1993.

[2] Chu-Cheow Lim and A. Stolcke. Sather language design and performance evaluation. Technical Report TR-91-034, International Computer Science Institute, Berkeley, May 1991.

[3] Scott Milton and Heinz W. Schmidt. Dynamic dispatch in object-oriented languages. Technical Report TR-CS-94-02, CSIRO – Division of Information Technology, Canberra, Australia, January 1992.

[4] Stephan Murer, Stephen Omohundro, and Clemens Szyperski. Sather Iters: Object-oriented iteration abstraction. Technical Report TR-93-045, International Computer Science Institute, Berkeley, August 1993.

[5] Object-Orientation FAQ. http://iamwww.unibe.ch/ scg/OOinfo/FAQ.

[6] Stephen M. Omohundro. The differences between Sather and Eiffel. *Eiffel Outlook*, 1(1):12–14, April 1991.

[7] Stephen M. Omohundro. Sather's design. *Eiffel Outlook*, 1(3):20–21, August 1991.

[8] Stephen M. Omohundro. Sather provides nonproprietary access to object-oriented programming. *Computer in Physics*, 6(5):444–449, September 1992.

[9] Stephen M. Omohundro. The Sather programming language. *Dr. Dobb's Journal*, 18(11):42–48, October 1993.

[10] Stephen M. Omohundro. The Sather 1.0 specification. Technical Report TR-in preparation, International Computer Science Institute, Berkeley, 1994.

[11] Stephen M. Omohundro and Chu-Cheow Lim. The Sather language and libraries. Technical Report TR-92-017, International Computer Science Institute, Berkeley, March 1992.

[12] Heinz W. Schmidt and Stephen M. Omohundro. CLOS, Eiffel, and Sather: A comparison. In Andreas Paepcke, editor, *Object-Oriented Programming: The CLOS Perspective*, pages 181–213. MIT Press Cambridge, Massachusetts, London, England, 1993. Available as ICSI TR-91-047.

[13] Clemens Szyperski, Stephen Omohundro, and Stephan Murer. Engineering a programming language: The type and class system of Sather. In Jurg Gutknecht, editor, *Programming Languages and System Architectures*, pages 208–227. Springer Verlag, Lecture Notes in Computer Science 782, November 1993. Available as technical report ICSI TR-93-064.