



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	1
1.1	Where can I find Sather?	1
1.2	Where can I read about Sather?	1
1.3	Related Work: Sather-K	2
1.4	Planned Changes to this Tutorial	2
2	Sather Tutorial Chess	3
2.1	Hello World Program	3
2.2	Getting Started	4
2.3	Class Hierarchy of Sather Tutorial Chess	4
3	Class Main	5
3.1	Routine main	6
3.2	Routine setup	7
4	Type \$CHESS_DISPLAY and Related Classes	11
4.1	Type \$CHESS_DISPLAY	11
4.2	Class CHESS_DISPLAY	12
4.3	Class ASCIILDISPLAY	14
4.4	Class X_DISPLAY	16
4.5	External Class XCW	17
4.6	Class DEFAULT	18
5	Type \$PLAYER and Related Classes	20
5.1	\$PLAYER	20
5.2	Class PLAYER	20
5.3	Class HUMAN_PLAYER	21
5.4	Class MINMAX	21
6	Class MOVE	25
7	Class POS	28
8	Class BOARD	38
9	Type \$PIECE and Related Classes	48
9.1	Type \$PIECE	48
9.2	Class PIECE	48
9.3	Class BISHOP	49
9.4	Class ROOK	50
9.5	Class QUEEN	51
9.6	Class KNIGHT	51
9.7	Class PAWN	52
9.8	Class KING	54
10	Suggested Exercises	57
	References	58

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      1
-- implemented in Sather 1.0                    2
class MAIN is                                    3
  main is                                        4
    #OUT + "Hello World\n";                      5
  end;                                           6
end;                                             7
```

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 1
  shared board : BOARD; 2
  shared display : $CHESS_DISPLAY; 3
  shared white, black, player : $PLAYER; 4
```

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 **structs** and reference types are pointers to **structs**.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 tutorial².

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.

```

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

```

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.

¹Furthermore, you are not allowed to have pointers directly to fields of **structs**.

²Some 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.

```

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

```

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.

```

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

```

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 `#` symbol 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.

```

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

```

```

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

```

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.

```

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

```

```
#OUT+"          or    C for Computer Player\n";          93
#OUT+"          <black> dito\n";                        94
#OUT+"          <Displ> can be either X for X Interface\n"; 95
#OUT+"          or    A for ASCII Terminal\n";          96
end;                                                    97
-- Since setup has a return parameter, a result        98
-- has to be returned to the caller.                  99
return ret;                                           100
end; -- of setup                                       101
end; -- of class MAIN                                  102
```

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.)

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

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.

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

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.

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

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**.

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

```

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

```

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

```

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

```

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 user 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.

```

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

```

end;	72
end; -- of <i>ask_pawn_xchg</i>	73
-- The following routine is included unchanged in <i>ASCII_DISPLAY</i>	74
-- and replaced in <i>X_DISPLAY</i> .	75
close is	76
end;	77
end; -- of <i>CHES_DISPLAY</i>	78

4.3 Class ASCII_DISPLAY

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

ASCII_DISPLAY inherits the implementation of class CHES_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 CHES_DISPLAY, the routine **create** does not return an object of type CHES_DISPLAY after being included in ASCII_DISPLAY. Instead, **create** returns an object of type ASCII_DISPLAY.

class ASCII_DISPLAY < \$CHES_DISPLAY is	79
include CHES_DISPLAY;	80

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.

```

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

```

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.

```

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

```

```

end; -- of getmove                                     126
private showtext(text:STR) is                          127
  #OUT+text+"\n";                                     128
end;                                                  129
end; -- of ASCII_DISPLAY                               130

```

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                    1
  include CHESS_DISPLAY;                               2
  create:SAME is                                      3
    XCW::OpenCW("Sather Tutorial Chess");             4
    return new;                                       5
  end;                                                6
  redraw(board:ARRAY{CHAR}) is                         7
    XCW::RedrawCW(board);                             8
  end;                                                9
  update(board:ARRAY{CHAR}) is                       10
    XCW::UpdateCW(board);                            11
  end;                                                12
  showmove(text:STR) is                               13
    XCW::ShowMoveCW(text);                           14
  end;                                                15
  private showtext(text:STR) is                       16
    XCW::TextCW(text);                               17
  end;                                                18

```

```

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

```

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`.

```

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

```

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                                     49
  OpenCW(title:STR);                                     50
  RedrawCW(board:ARRAY{CHAR});                           51
  UpdateCW(board:ARRAY{CHAR});                           52
  GetMoveInCW(move:ARRAY{CHAR});                         53
  ShowMoveCW(move:STR);                                  54
  TextCW(text:STR);                                     55
  CloseCW;                                              56
end;                                                    57

```

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                                       1
  display(d:CHAR):$CHESS_DISPLAY is                     2
    ret : $CHESS_DISPLAY;                                3
    if d = 'X' then                                     4
      -- Create an object of type X_DISPLAY and return it. 5
      -- To be more specific: # is a short-hand for a call to 6
      -- the the routine create of type that follows the #. 7
      ret := #X_DISPLAY;                                 8
    else                                                9
      ret := #ASCII_DISPLAY;                             10
    end;                                               11
  return ret;                                         12

```

```
end; 13
end; 14
```

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 1
  display(d:CHAR):$CHESS_DISPLAY is 2
    ret : $CHESS_DISPLAY; 3
    -- Since X is not available, create ASCII-Interface only. 4
    ret := #ASCII_DISPLAY; 5
    return ret; 6
  end; 7
end; 8
```

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                                     1
  getmove(b:BOARD):MOVE;                           2
  ask_pawn_xchg:CHAR;                               3
end;                                                4
```

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                          5
  attr iswhite:BOOL;                                6
  create(iswhite:BOOL):SAME is                      7
    ret : SAME := new;                               8
    ret.iswhite := iswhite;                          9
    return ret;                                     10
  end;                                              11
  getmove(b:BOARD):MOVE is                          12
    raise "PLAYER:invalid call to getmove\n";       13
  end;                                              14
  ask_pawn_xchg:CHAR is                             15
    return 'Q';                                     16
  end;                                              17
end; -- of class PLAYER                             18
```

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                                     19
  include PLAYER;                                          20
  getmove(b:BOARD):MOVE is                                 21
    return MAIN::display.getmove(iswhite);                22
  end;                                                    23
  ask_pawn_xchg:CHAR is                                    24
    MAIN::display.update(MAIN::board.str);                25
    return MAIN::display.ask_pawn_xchg;                    26
  end;                                                    27
end; -- of class HUMAN                                     28
```

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.

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

```

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

```

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.

```

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

```

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`.

```

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

```

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.

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

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.

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

```

ret.king_chg := false;                                18
ret.pawn_chg := false;                                19
if void(move) then return ret; end;                  20
if move.size >= 5 and move.head(5) = "o-o-o" then    21
  ret.from := #POS; ret.to := #POS;                  22
  ret.isq_castle := true;                             23
  if white_to_move then                                24
    ret.from.pos := "e1"; ret.to.pos := "c1";          25
  else                                                    26
    ret.from.pos := "e8"; ret.to.pos := "c8";          27
  end;                                                    28
elsif move.size >= 3 and move.head(3) = "o-o" then  29
  ret.from := #POS; ret.to := #POS;                  30
  ret.isk_castle := true;                             31
  if white_to_move then                                32
    ret.from.pos := "e1"; ret.to.pos := "g1";          33
  else                                                    34
    ret.from.pos := "e8"; ret.to.pos := "g8";          35
  end;                                                    36
elsif move.size >= 4 and move.head(4) = "quit" then 37
  ret.isquit := true;                                   38
elsif move.size >= 5 then                               39
  str_from := move.substring(0,2);                    40
  if POS::check_pos(str_from) then                    41
    ret.from := #POS; ret.from.pos := str_from;        42
  end;                                                    43
  str_to := move.substring(3,2);                      44
  if POS::check_pos(str_to) then                    45
    ret.to := #POS; ret.to.pos := str_to;              46
  end;                                                    47
elsif move.size >=4 then                               48
  str_from := move.substring(0,2);                    49
  if POS::check_pos(str_from) then                    50
    ret.from := #POS; ret.from.pos := str_from;        51
  end;                                                    52
  str_to := move.substring(2,2);                      53
  if POS::check_pos(str_to) then                    54
    ret.to := #POS; ret.to.pos := str_to;              55
  end;                                                    56
end;                                                    57
return ret;                                            58
end; -- of first version of create                    59

```

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 60
  ret := new; 61
  ret.isk_castle := false; 62
  ret.isq_castle := false; 63
  ret.isquit := false; 64
  ret.from := #POS; 65
  ret.from.pos := piece.position.str; 66
  ret.to := #POS; 67
  ret.to.pos := to.str; 68
  ret.piece := void; 69
  ret.kills := void; 70
  ret.king_chg := false; 71
  ret.pawn_chg := false; 72
  if piece.isking then 73
    if piece.iswhite then 74
      if piece.position = "e1" and to = "c1" then 75
        ret.isq_castle := true; 76
      end; 77
      if piece.position = "e1" and to = "g1" then 78
        ret.isk_castle := true; 79
      end; 80
    else 81
      if piece.position = "e8" and to = "c8" then 82
        ret.isq_castle := true; 83
      end; 84
      if piece.position = "e8" and to = "g8" then 85
        ret.isk_castle := true; 86
      end; 87
    end; 88
  end; 89
  return ret; 90
end; -- of second version of create 91
isok:BOOL is 92
  return ~void(from) and ~void(to); 93
end; 94
end; -- of class MOVE 95

```

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.

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

```

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

```

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

```

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

```

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 syntax error is that there may not be a semicolon behind a pre-condition because it is part of the header.

```

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

```

```

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

```

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.

```

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

```



```

    when 2 then return 'c';           71
    when 3 then return 'd';           72
    when 4 then return 'e';           73
    when 5 then return 'f';           74
    when 6 then return 'g';           75
    when 7 then return 'h';           76
end;                                   77
end;                                   78
private str(pos:INT):STR is           79
    ret := #STR;                       80
    col := pos % 8;                     81
    row := (pos / 8) + 1;               82
    case col                             83
        when 0 then ret := "a";         84
        when 1 then ret := "b";         85
        when 2 then ret := "c";         86
        when 3 then ret := "d";         87
        when 4 then ret := "e";         88
        when 5 then ret := "f";         89
        when 6 then ret := "g";         90
        when 7 then ret := "h";         91
    end;                                 92
    ret := ret + row;                   93
    return ret;                         94
end; -- of str(INT)                    95

```

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                             96
    ret := absolute + 1;                 97
    if ret/8 /= absolute/8 then ret := absolute; end; 98
    return str(ret);                     99
end;                                     100
west:STR is                              101
    ret := absolute - 1;                 102
    if ret/8 /= absolute/8 then ret := absolute; end; 103
    return str(ret);                     104
end;                                     105
north:STR is                              106
    ret := absolute + 8;                 107
    if ret > 63 then ret := absolute; end; 108
    return str(ret);                     109
end;                                     110
south:STR is                              111
    ret := absolute - 8;                 112
    if ret < 0 then ret := absolute; end; 113
    return str(ret);                     114
end;                                     115

```

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	116
err : BOOL := false;	117
ret := absolute + 8;	118
if ret > 63 then ret := absolute; err := true; end;	119
if ~err then	120
ret := absolute + 1;	121
if ret/8 /= absolute/8 then ret := absolute; err := true; end;	122
end;	123
if ~err then	124
ret := absolute + 9;	125
end;	126
return str(ret);	127
end;	128
northwest:STR is	129
err : BOOL := false;	130
ret := absolute + 8;	131
if ret > 63 then ret := absolute; err := true; end;	132
if ~err then	133
ret := absolute - 1;	134
if ret/8 /= absolute/8 then ret := absolute; err := true; end;	135
end;	136
if ~err then	137
ret := absolute + 7;	138
end;	139
return str(ret);	140
end;	141
southeast:STR is	142
err : BOOL := false;	143
ret := absolute - 8;	144
if ret < 0 then ret := absolute; err := true; end;	145
if ~err then	146
ret := absolute + 1;	147
if ret/8 /= absolute/8 then ret := absolute; err := true; end;	148
end;	149
if ~err then	150
ret := absolute - 7;	151
end;	152
return str(ret);	153
end;	154
southwest:STR is	155
err : BOOL := false;	156
ret := absolute - 8;	157
if ret < 0 then ret := absolute; err := true; end;	158
if ~err then	159
ret := absolute - 1;	160
if ret/8 /= absolute/8 then ret := absolute; err := true; end;	161
end;	162
if ~err then	163

```

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

```

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.

```

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

```

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, an 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 evaluated³.

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

³An 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.

```

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

```

```

        quit; 220
    end; 221
else 222
    quit; 223
end; 224
when diag_dn_left then 225
    if stepped.column > 'a' then 226
        if stepped.row < '8' then 227
            stepped.pos := stepped.northwest; 228
            ret := #POS; 229
            ret.pos := stepped.pos; 230
            yield ret; 231
        else 232
            quit; 233
        end; 234
    else 235
        quit; 236
    end; 237
when vertical_up then 238
    if stepped.row < '8' then 239
        stepped.pos := stepped.north; 240
        ret := #POS; 241
        ret.pos := stepped.pos; 242
        yield ret; 243
    else 244
        quit; 245
    end; 246
when vertical_dn then 247
    if stepped.row > '1' then 248
        stepped.pos := stepped.south; 249
        ret := #POS; 250
        ret.pos := stepped.pos; 251
        yield ret; 252
    else 253
        quit; 254
    end; 255
when horizontal_right then 256
    if stepped.column < 'h' then 257
        stepped.pos := stepped.east; 258
        ret := #POS; 259
        ret.pos := stepped.pos; 260
        yield ret; 261
    else 262
        quit; 263
    end; 264
when horizontal_left then 265
    if stepped.column > 'a' then 266
        stepped.pos := stepped.west; 267
        ret := #POS; 268
        ret.pos := stepped.pos; 269
        yield ret; 270

```

else	271
quit ;	272
end ; -- <i>way! will be continued ...</i>	273

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

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

```

    ret.pos := stepped.pos;          312
    count := count + 1;             313
    yield ret;                       314
    if row /= '7' then quit; end;    315
else                                  316
    quit;                             317
end;                                  318
when ring then                        319
    ret := #POS;                     320
    case count                        321
        when 0 then ret.pos := north; 322
        when 1 then ret.pos := south;  323
        when 2 then ret.pos := east;   324
        when 3 then ret.pos := west;   325
        when 4 then ret.pos := northeast; 326
        when 5 then ret.pos := northwest; 327
        when 6 then ret.pos := southeast; 328
        when 7 then ret.pos := southwest; 329
    else                               330
        quit;                           331
    end;                                332
    count := count + 1;                333
    if ret.pos <= 63 and ret.pos >= 0  334
        and ret.pos /= absolute        335
        and column(ret) <= column(self) + 1 336
        and column(self) - 1 <= column(ret) 337
        and row(ret) <= row(self) + 1      338
        and row(self) - 1 <= row(ret)      339
    then                                340
        yield ret;                      341
    end;                                342
else                                    343
    -- The else case was put in for reasons of
    -- fail safe program development.      344
    raise "POS:way! invalid case\n";      345
end; -- of case                         346
end; -- of loop                         347
end; -- of way!                         348
end; -- of class POS                    349
end; -- of class POS                    350

```

8 Class BOARD

The two array `whitepieces` 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 1
  private attr whitepieces : ARRAY{$PIECE}; 2
  private attr blackpieces : ARRAY{$PIECE}; 3
    attr white_to_play : BOOL; 4
    attr last_move : MOVE; 5
    attr white_K_moved : BOOL; 6
    attr black_K_moved : BOOL; 7
  create:SAME is 8
    ret::=new; 9
    ret.set_up; 10
    return ret; 11
  end; 12
```

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 13
  position := #POS; 14
  white_to_play := true; 15
  -- set up white pieces 16
  whitepieces := #(16); 17
  position.pos := "a2"; 18
  loop i:=0 upto!(7); 19
    whitepieces[i] := #PAWN(position,PIECE::white); 20
    position.pos := position.east; 21
  end; 22
  position.pos := "a1"; whitepieces[8] := #ROOK(position,PIECE::white); 23
  position.pos := "b1"; whitepieces[9] := #KNIGHT(position,PIECE::white); 24
  position.pos := "c1"; whitepieces[10] := #BISHOP(position,PIECE::white); 25
  position.pos := "d1"; whitepieces[11] := #QUEEN(position,PIECE::white); 26
  position.pos := "e1"; whitepieces[12] := #KING(position,PIECE::white); 27
  position.pos := "f1"; whitepieces[13] := #BISHOP(position,PIECE::white); 28
  position.pos := "g1"; whitepieces[14] := #KNIGHT(position,PIECE::white); 29
  position.pos := "h1"; whitepieces[15] := #ROOK(position,PIECE::white); 30
  -- set up black pieces 31
  blackpieces := #(16); 32
  position.pos := "a7"; 33
  loop i:=0 upto!(7); 34
    blackpieces[i] := #PAWN(position,PIECE::black); 35
    position.pos := position.east; 36
```



```

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

```

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.

```

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

```

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.

```

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

```

```

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

```

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.)

```

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

```

ret := i;	101
break!;	102
end;	103
end; -- of loop	104
else	105
ret := blackpieces.index_of(p);	106
end;	107
return ret;	108
end; -- private index	109

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	110
ret : BOOL := false;	111
loop p::=piece!;	112
if p.position = pos then ret := true; end;	113
end;	114
return ret;	115
end;	116
has_white_piece(pos:POS):BOOL is	117
ret : BOOL := false;	118
loop p::=whitepiece!;	119
if p.position = pos then ret := true; end;	120
end;	121
return ret;	122
end;	123
has_black_piece(pos:POS):BOOL is	124
ret : BOOL := false;	125
loop p::=blackpiece!;	126
if p.position = pos then ret := true; end;	127
end;	128
return ret;	129
end;	130
has_my_piece(pos:POS):BOOL is	131
if white_to_play then	132
return has_white_piece(pos);	133
else	134
return has_black_piece(pos);	135
end;	136
end;	137

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	138
ret : \$PIECE;	139
position ::= #POS;	140
position.pos := str;	141
loop p::=piece!;	142

```

    if p.position = position then ret := p; end;      143
end;                                                144
return ret;                                        145
end;                                                146
piece(p:POS):$PIECE is                             147
    return piece(p.str);                           148
end;                                                149

```

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                                 150
ret:=#ARRAY{CHAR}(65);                             151
loop                                                152
    ret[0.upto!(63)] := ' ';                         153
end;                                                154
ret[64] := '\0';                                    155
loop p:=self.whitepiece!;                           156
    if ~void(p) and p.alive then                     157
        ret[p.position.pos] := p.fig;                 158
    end;                                             159
end;                                                160
loop p:=self.blackpiece!;                            161
    if ~void(p) and p.alive then                     162
        ret[p.position.pos] := p.fig.lower;           163
    end;                                             164
end;                                                165
return ret;                                         166
end;                                                167

```

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 174 is a dispatched iter. See page 24 for an alternative implementation that works with earlier releases of the Sather 1.0 compiler.

```

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

```

```

        break!;                                177
    end;                                        178
end;                                           179
if ret then break!; end;                       180
end;                                           181
return ret;                                    182
end; -- of pos_in_check                         183

```

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                    184
    piece : $PIECE;                            185
    loop                                       186
        piece := my_piece!;                    187
        until!(piece.isking);                 188
    end;                                       189
    return pos_in_check(piece.position);       190
end; -- of my_king_isin_check                 191

```

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        192
    return (move_valid_so_far(move) and apply_move_with_own_check_test(move)); 193
end; -- of check_n_apply_move                 194
private move_valid_so_far(move:MOVE):BOOL    195
pre ~ move.isquit                             196
is                                             197
    ret : BOOL := false;                       198
    -- A valid move must start at a position where one of my pieces is... 199
    if has_my_piece(move.from) then           200
        p:=piece(move.from);                  201
        -- ... and it must be a valid move with respect to the mobility of the 202
        -- piece at the current state of the board.                             203
        if p.valid_move(move.to,self) then    204
            ret := true;                       205
            -- Since the move seems to be valid, the moving piece is stored      206
            -- in the move object. That eases future access to the moving piece 207
            -- and allows for un-doing of moves.                                   208
            move.piece := p;                   209
        end;                                   210
    end;                                       211
return ret;                                    212

```

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.

```

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

```

```

    end;
236
end;
237
if ~void(r) then
238
    move.kills := r;
239
    r.alive := false;
240
end;
241
p.update_position(move.to);
242
-- Deal with king moves.
243
if p.isking then
244
    if white_to_play and ~white_K_moved then
245
        white_K_moved := true;
246
        move.king_chg := true;
247
    end;
248
    if ~white_to_play and ~black_K_moved then
249
        black_K_moved := true;
250
        move.king_chg := true;
251
    end;
252
end;
253
-- Deal with pawn exchange.
254
if (p.ispawn and p.iswhite and white_to_play and p.position.row='8')
255
    or (p.ispawn and ~p.iswhite and ~white_to_play and p.position.row='1')
256
then
257
    case MAIN::player.ask_pawn_xchg
258
    when 'Q' then
259
        whitepieces[index(p)] := #QUEEN(p.position,PIECE::white);
260
    when 'K' then
261
        whitepieces[index(p)] := #KNIGHT(p.position,PIECE::white);
262
    else
263
        -- Do it fails safe.
264
        raise "BOARD:apply_move:pawn_exchange invalid case entry\n"
265
    end;
266
    move.pawn_chg := true;
267
end;
268
-- Deal with castles.
269
if move.isq_castle then
270
    if white_to_play then
271
        rook ::= piece("a1");
272
        rook.update_position("d1");
273
    else
274
        rook ::= piece("a8");
275
        rook.update_position("d8");
276
    end;
277
elseif move.isk_castle then
278
    if white_to_play then
279
        rook ::= piece("h1");
280
        rook.update_position("f1");
281
    else
282
        rook ::= piece("h8");
283
        rook.update_position("f8");
284
    end;
285
end;
286

```

```

move.prev_move := last_move;                                287
last_move := move;                                        288
-- Check whether my king is in check after application of the move 289
if my_king_isin_check then                                290
  -- Although otherwise correct this is an invalid move.        291
  -- The original state of the board is reconstructed by calling 292
  -- unapply_move.                                              293
  ret := false;                                           294
  unapply_move;                                             295
end;                                                       296
white_to_play := ~white_to_play;                          297
return ret;                                               298
end; -- of apply_move                                     299

```

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                                           300
  -- Restore killed opponent piece                            301
  if ~void(last_move.kills) then                            302
    last_move.kills.alive := true;                          303
  end;                                                       304
  -- Restore pawn exchange                                    305
  if last_move.pawn_chg then                                 306
    newpiece ::= piece(last_move.piece.position);           307
    whitepieces[index(newpiece)] := last_move.piece;        308
  end;                                                       309
  -- Restore move                                            310
  last_move.piece.update_position(last_move.from);          311
  if last_move.king_chg then                                312
    if white_to_play then                                    313
      white_K_moved := false;                               314
    else                                                      315
      black_K_moved := false;                               316
    end;                                                       317
  end;                                                       318
  -- Restore castle                                          319
  if last_move.isq_castle then                               320
    if white_to_play then                                    321
      rook ::= piece("d1");                                  322
      rook.update_position("a1");                            323
    else                                                      324
      rook ::= piece("d8");                                  325
      rook.update_position("a8");                            326
    end;                                                       327
  elsif last_move.isk_castle then                            328
    if white_to_play then                                    329
      rook ::= piece("f1");                                  330
      rook.update_position("h1");                            331

```



```

else                                                    332
  rook ::= piece("f8");                               333
  rook.update_position("h8");                         334
end;                                                  335
end;                                                  336
last_move := last_move.prev_move;                   337
white_to_play := ~white_to_play;                    338
end; -- of unapply_move                               339

```

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                                    340
  white_value : INT := 0;                             341
  black_value : INT := 0;                             342
  loop p ::= whitepiece!;                             343
    white_value := white_value + p.worth;             344
  end;                                                345
  loop p ::= blackpiece!;                             346
    black_value := black_value + p.worth;             347
  end;                                                348
  return white_value.flt/black_value.flt;            349
end; -- of board_value                               350
end; -- of class BOARD                               351

```

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 1
  alive:BOOL; 2
  alive(set:BOOL); 3
  worth:INT; 4
  iswhite:BOOL; 5
  position:POS; 6
  valid_move(to:POS,board:BOARD):BOOL; 7
  update_position(position:POS); 8
  update_position(position:STR); 9
  move!(b:BOARD,to_piece:BOOL):POS; 10
  fig:CHAR; 11
  ispawn : BOOL; 12
  isrook : BOOL; 13
  isking : BOOL; 14
end; -- of type $PIECE 15
```

9.2 Class PIECE

```
class PIECE < $PIECE is 16
  -- General constants that are used throughout the descendants of $PIECE 17
  const white : BOOL := true; 18
  const black : BOOL := ~white; 19
  const ordinary : BOOL := false; 20
  const for_check_test : BOOL := true; -- alters behavior of move! 21
  -- Attributes that are specific to a PIECE 22
  attr alive : BOOL; 23
  attr iswhite : BOOL; 24
  attr position : POS; 25
  -- Constants that are specific to a PIECE 26
  const worth : INT := 0; 27
  const fig : CHAR := ' '; 28
  const ispawn : BOOL := false; 29
  const isking : BOOL := false; 30
  const isrook : BOOL := false; 31
  create(pos:POS,iswhite:BOOL):SAME is 32
    ret := new; 33
    ret.position := #POS; 34
    ret.position.pos := pos.str; 35
    ret.iswhite := iswhite; 36
```

```

    ret.alive := true;                                     37
    return ret;                                          38
end;                                                    39
private same_color(b:BOARD,p:POS):BOOL                40
pre b.has_piece(p)                                     41
is                                                     42
    white_piece_on_pos :BOOL:= b.has_white_piece(p);  43
    if ( iswhite and white_piece_on_pos)              44
        or (~iswhite and ~white_piece_on_pos) then  45
            return true;                               46
        else                                           47
            return false;                              48
        end;                                           49
    end;                                               50

```

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.

```

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

```

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                                     68
  include PIECE;                                           69
  -- Constants that are different from PIECE implementation: 70
  const worth : INT := 3;                                   71
  const fig : CHAR := 'B';                                  72
  move!(b:BOARD,mode:BOOL):POS is                           73
    to : POS;                                              74
    loop direction::=POS::diag_up_right.upto!(POS::diag_dn_left); 75
      loop to:=position.way!(direction);                    76
        if ~b.has_piece(to) then                            77
          yield to;                                         78
          elsif same_color(b,to) and mode=ordinary then    79
            break!;                                         80
          else                                              81
            yield to;                                       82
            break!;                                         83
          end;                                              84
        end;                                              85
      end;                                              86
    end; -- of move!                                       87
end; -- of class BISHOP                                     88

```

9.4 Class ROOK

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

```

class ROOK < $PIECE is                                     89
  include PIECE;                                           90
  -- Constants that are different from PIECE implementation: 91
  const worth : INT := 5;                                   92
  const fig : CHAR := 'R';                                  93
  const isrook : BOOL := true;                              94
  move!(b:BOARD,mode:BOOL):POS is                           95
    -- returns all valid moves given a board with other pieces 96
    to : POS;                                              97
    -- This loop will check the following directions:        98
    -- horizontal_right, horizontal_left, vertical_up, vertical_dn 99
    loop direction::=POS::horizontal_right.upto!(POS::vertical_dn); 100
      loop to:=position.way!(direction);                    101
        if ~b.has_piece(to) then                            102
          yield to;                                         103
          elsif same_color(b,to) and mode=ordinary then    104
            break!;                                         105
          else                                              106
            yield to;                                       107
            break!;                                         108
          end;                                              109
        end;                                              110
      end; -- of move!                                       111
    end;

```

9.5 Class QUEEN

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

```

class QUEEN < $PIECE is                                     113
  include PIECE;                                          114
  -- Constants that are different from PIECE implementation: 115
  const worth : INT := 9;                                116
  const fig : CHAR := 'Q';                                117
  move!(b:BOARD,mode:BOOL):POS is                          118
    -- returns all valid moves given a board with other pieces 119
    to : POS;                                             120
    -- This loop will check the following directions:          121
    -- diag_up_right, diag_up_left, diag_dn_right, diag_dn_left 122
    -- horizontal_right, horizontal_left, vertical_up, vertical_dn 123
    -- It is a combination of rook and bishop.                124
    loop direction:=POS::diag_up_right.upto!(POS::vertical_dn); 125
      loop to:=position.way!(direction);                    126
        if ~b.has_piece(to) then                            127
          yield to;                                         128
        elsif same_color(b,to) and mode = ordinary then break!; 129
        else                                                130
          yield to;                                         131
          break!                                           132
        end;                                                133
      end;                                                  134
    end;                                                    135
  end; -- of move!                                         136
end; -- of class QUEEN                                    137

```

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                                   138
  include PIECE;                                          139
  -- Constants that are different from PIECE implementation: 140
  const worth : INT := 3;                                141
  const fig : CHAR := 'N';                                142
  move!(b:BOARD,mode:BOOL):POS is                          143
    -- returns all valid moves given a board with other pieces 144
    to : POS;                                             145
    loop to:=position.way!(POS::knight);                  146
      if b.has_piece(to) and same_color(b,to) and mode = ordinary then 147

```

```

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

```

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.

```

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

```

```

to.pos := position.northwest; 189
if mode = for_check_test then 190
  yield to; 191
else 192
  if b.has_black_piece(to) then 193
    yield to; 194
  end; 195
end; 196
end; 197
-- en passant 198
if position.row = '5' 199
  and ~void(b.last_move) 200
  and b.last_move.from.row = '7' 201
  and ( b.last_move.to = position.east 202
        or b.last_move.to = position.west) 203
  and ~void(b.last_move.piece) and b.last_move.piece.ispawn 204
then 205
  if mode = for_check_test then 206
    yield b.last_move.to; 207
  else 208
    to := #POS; 209
    to.pos := b.last_move.to.north; 210
    yield to; 211
  end; 212
end; 213
-- no more moves; 214
quit; 215
else -- i.e. if isblack 216
  if mode = ordinary then 217
    -- vertical steps 218
    loop to:=position.way!(POS::south_two); 219
      if b.has_piece(to) then -- position and continued way blocked 220
        break!; 221
      end; 222
      yield to; 223
    end; 224
  end; 225
  -- diag_up 226
  if position.column > 'a' then 227
    to:=#POS; 228
    to.pos := position.southwest; 229
    if mode = for_check_test then 230
      yield to; 231
    else 232
      if b.has_white_piece(to) then 233
        yield to; 234
      end; 235
    end; 236
  end; 237
  -- diag_dn 238
  if position.column < 'h' then 239

```

```

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

```

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.

```

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

```



```

if b.has_piece(to) and same_color(b,to) and mode = ordinary then      281
  -- skip this move                                                    282
else                                                                      283
  if mode = for_check_test or ~b.pos_in_check(to) then                284
    yield to;                                                            285
  end;                                                                    286
end;                                                                      287
end;                                                                      288
-- castle moves                                                         289
spot1, spot2, spot3, rook : $PIECE;                                       290
if b.white_to_play and ~b.white_K_moved and position = "e1" then      291
  -- q-castle                                                            292
  spot1:= b.piece("d1"); spot2:= b.piece("c1"); spot3:= b.piece("b1");  293
  rook := b.piece("a1");                                                 294
  if ~void(rook) and rook.isrook and rook.alive                       295
    and void(spot1) and void(spot2) and void(spot3)                   296
  then                                                                    297
    to := #POS;                                                         298
    to.pos := "d1";                                                     299
    if ~b.pos_in_check(to) then                                         300
      to.pos := "c1";                                                  301
      if ~b.pos_in_check(to) then                                       302
        yield to;                                                       303
      end;                                                                304
    end;                                                                305
  end;                                                                    306
-- k-castle                                                             307
spot1:= b.piece("f1"); spot2:= b.piece("g1"); rook := b.piece("h1");  308
if ~void(rook) and rook.isrook and rook.alive                       309
  and void(spot1) and void(spot2)                                       310
then                                                                    311
  to := #POS;                                                         312
  to.pos := "f1";                                                     313
  if ~b.pos_in_check(to) then                                         314
    to.pos := "g1";                                                  315
    if ~b.pos_in_check(to) then                                       316
      yield to;                                                       317
    end;                                                                318
  end;                                                                319
end;                                                                    320
end; -- castle moves of white king                                       321
if ~b.white_to_play and ~b.black_K_moved and position = "e8" then    322
  -- q-castle                                                            323
  spot1:= b.piece("d8"); spot2:= b.piece("c8"); spot3:= b.piece("b8");  324
  rook := b.piece("a8");                                                 325
  if ~void(rook) and rook.isrook and rook.alive                       326
    and void(spot1) and void(spot2) and void(spot3)                   327
  then                                                                    328
    to := #POS;                                                         329
    to.pos := "d8";                                                  330
    if ~b.pos_in_check(to) then                                       331

```

```

    to.pos := "c8";
    if ~b.pos_in_check(to) then
        yield to;
    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 Exercises

- 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 Colorado, 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.