

# Dialaw: the program

Arno Lodder and Paul Huygen

December 8, 1999

## Contents

1	<b>Introduction</b>	<b>2</b>
1.1	<i>Literate programming</i>	2
2	<b>The main structures of the program</b>	<b>3</b>
2.1	<i>The data-types move, dialog and commitment store</i>	3
3	<b>The main program loop</b>	<b>3</b>
3.1	<i>first_ move</i>	3 4
3.2	<i>dialog</i>	3 4
4	<b>The moves</b>	<b>6</b>
4.1	<i>The forced moves</i>	6
4.2	<i>General move input (voluntary, non-forced moves)</i>	10
4.2.1	<i>Response to a claim</i>	11
4.2.2	<i>Response to a question</i>	11
4.2.3	<i>Response to accept and withdrawal</i>	12
4.2.4	<i>Find the level of a claim</i>	15
4.3	<i>Check the validity of moves</i>	15
5	<b>The commitment store</b>	<b>20</b>
5.1	<i>Update the commitment store</i>	20
5.1.1	<i>Consult the commitment store</i>	22
6	<b>Other elements of Dialaw</b>	<b>22</b>
6.1	<i>Change the players</i>	22
6.2	<i>previous withdrawal</i>	22
6.3	<i>The move counter</i>	23

6.4	<i>Some code that I don't understand</i>	23
7	<b>Input and output</b>	<b>23</b>
7.1	<i>Low level input from/output to the terminal</i>	23
7.2	<i>Windows management and user interface</i>	24
8	<b>Put everything in a Prolog file</b>	<b>28</b>
9	<b>Indexes</b>	<b>30</b>

## 1 Introduction

This document contains the program code of a program called *Dialaw*. This program models judicial dialogs. A description of the program can be found in [3]. The program code in this document deviates from the original listing in [3], because it has been adapted to another software platform.

*Dialaw* has been written in the Prolog programming language. In order to facilitate a comfortable user interface, a Prolog dialect that provides tools for such an interface has been used. However, care has been taken to separate the platform-dependent code as good as possible from the code that uses the standard language, in order to facilitate porting to other platforms.

The code and the explanations are best understood if one is acquainted with Prolog or a similar declarative language. In the text the number of objects are often replaced with a number indicating its arity. For instance, the structure `date(Year, Month, Day)` would become `date/3`.

The original *Dialaw*, as listed in [3], was written for a Prolog interpreter called *Prolog 2 for Windows*. However, this Prolog dialect is no longer supported by its manufacturer. Therefore, in this version another software platform was used, i.e. SWI-Prolog, extended with XPCE. Both products have been developed by the department of Social Sciences Informatics (SWI) of the University of Amsterdam (Roetersstraat 15 1018WB Amsterdam, the Netherlands). They can be downloaded from their Internet site [www.swi.psy.uva.nl](http://www.swi.psy.uva.nl). SWI-Prolog is a free product. For XPCE a licence must be bought. However, without licence XPCE can be used to its full capabilities, but only for a limited time, after which it has to be restarted.

### 1.1 Literate programming

For this document the *Literate Programming* technique has been applied.

Literate programming is a methodology that combines a programming language with a documentation language, thereby making programs more robust, more portable and more easily maintained. The main idea is to treat a program as a piece of literature, addressed to human beings rather than to a computer. The Literate Programming concept was developed by D. Knuth [2].

Several literate programming systems exist, with different features, advantages and disadvantages. Generally they work as follows: An electronic document contains an alternation of

chunks of program code and text for the human reader. Generally there is a macro facility: a chunk of code can be replaced by a reference to a macro that is defined in some other place of the electronic document. This mechanism encourages a top down approach of programming and documenting. The literate programming system generally consists of two programs. One program strips the texts and substitutes the macro contents for the macro bodies (recursively), creating a source file that is readable for the compiler. The other program turns the electronic document into a document for a wordprocessor or typesetter.

This program/document has been written using the literate programming tool Funnelweb. The program and documentation are obtainable via Internet, e.g. from the following location:

<http://www.cs.ruu.nl/mirror/tex-archive/web/funnelweb/>

## 2 The main structures of the program

### 2.1 The data-types move, dialog and commitment store

The move, dialog, and commitment store are based on the definitions in chapter 3 of [3].

A move is stored in a variable `DialogMove` that refers to a structure with three elements, e.g. `(Player, (Act, Sentence), (Level, MoveAbout))`, where

**Player:** Reference to the player whose turn to move it is. The players are called either `bert` (`player1`) or `ernie` (`player2`);

**(Act, Sentence):** , the variable `Act` refers to the illocutionary act performed by the player of the dialog move, the variable `Sentence` refers to the propositional content of the illocutionary act; `Act` may have the following values: `claim`, `question`, `accept`, `withdraw` or `quit`.

**(Level, MoveAbout):** `Level` refers to the level the dialog move was made on an `MoveAbout` refers to the sentence the move is an argument for or reaction to.

The entire dialog is stored in a variable `DialogMoves` that refers to a Prolog-list with one or more elements of the type `DialogMove`. The number of elements in `Dialogmoves` is identical to the number of valid moves of the game. For instance, after three valid moves `DialogMoves` is a list with three elements.

The variable `CommitmentStore` refers to the Prolog list that contains the commitments of each player at a particular stage of the dialog. Suppose only the first player is committed to `murderer(oj)`, and both are committed to `shot_wife(oj)`. The commitment store would then be the following list:

```
[(bert, murderer(oj)),  
(bert, shot_wife(oj)), (ernie, shot_wife(oj))].
```

## 3 The main program loop

Typing `dialaw` starts the program. It calls the following Prolog-rule, which is the main clause of the program.

*the main clause*[1]  $\equiv$

```
{dialaw :-
    clean,
    create_windows,
    first_move(CS, FirstSentence, DialogMoves),
    dialog(CS, FirstSentence, DialogMoves),
    stop the program[75].}
```

This macro is invoked in definition 76.

Apart from some administrative chores, this rule defines under what circumstances `dialaw/0` is true, or, in other words, under what circumstances the program ends. This is the case if both `first_move/3` and `dialog/3` are true.

### 3.1 first\_move/3

The first move is always a claim of one of the players (call this player *player 1* or simply *Bert*). Therefore the illocutionary act of the first move is fixed as claim, and the player is allowed to enter the propositional content of the claim. The following clause represents the first move:

```
first_move(CommitmentStore, FirstSentence, DialogMoves)
```

When the player has entered the claim, `first_move/3` becomes true. After the first move `CommitmentStore` has a single element, i.e. the list containing the term `bert` and the text of the first claim. The list `DialogMoves` also contains only one element, recording that `Bert` has claimed a particular sentence on the zero level. Because his move is not a reaction to another move the variable `MoveAbout` is initiated as `dialaw` (see definition 5). Assume as an example that Bert claims in the first move the sentence `murderer(oj)`. In that case `first_move/3` becomes true with the following content:

```
first_move( [(bert, murderer(oj))]
            , murderer(oj)
            , [(bert, (claim, murderer(oj)), (0, dialaw))]
            )
```

Here comes the actual code of `first_move`.

```
the first move[2] ≡
{first_move([(bert, Sentence)], Sentence,
            [(bert, (claim, Sentence), (0, dialaw))]) :-
    set the move counter to unity[58]
    write the first status[67]
    ask Bert for Sentence[71],
    update the screen after the initial move[68].
}
```

This macro is invoked in definition 76.

### 3.2 dialog/3

The clause `dialog/3` has the same elements as that of the first move, namely: `dialog(CommitmentStore, FirstSentence, DialogMoves)`. After the first move, the program continues until `dialog/3` is true. There are basically two situations in which `dialog/3` becomes true, and hence the dialog ends. The first situation occurs, if after a move the set of

open sentences has become empty. This is the case when the first sentence has been accepted (by player 2), withdrawn (by player 1), or when the denial of the sentence has been accepted (by player 1 or withdrawn (by player 2). Each of those conditions make one of the following clauses true, thereby ending the game:

```

the conditions under which dialog/3 ends[3] + ≡
  {dialog(_, S, [(_, (accept, S), _)|_]).
   dialog(_, S, [(_, (withdraw, S), _)|_]).
   dialog(_, S, [(_, (accept, not(S)), _)|_]).
   dialog(CS, S, [(_, (withdraw, not(S)), _)|_]):-
     not_inCS((_, S), CS).
  }

```

This macro is defined in definitions 3 and 4.  
This macro is invoked in definition 76.

The second situation that causes the game to come to an end, occurs if a player has entered `quit` as an illocutionary act. The quit-option is built in because otherwise the program could not be stopped in the case that the players do not want to continue an unfinished dialog.

```

the conditions under which dialog/3 ends[4] + ≡
  {dialog(_, _, [(_, (quit, _), _)|_]).
  }

```

This macro is defined in definitions 3 and 4.  
This macro is invoked in definition 76.

In addition to the five clauses that define the circumstances under which `dialog/3` becomes true, there is a sixth `dialog/3` clause. This recursive clause runs in fact the whole of the dialog.

```

the main dialog clause[5] ≡
  {dialog(CS, FS, DM) :-
    move_input(CS, DM, NewPlayer, NewLevels, Move),
    trace,
    valid_move(NewLevels, NewPlayer, CS, DM, ValidMove, Move),
    notrace,
    clear_the_commitment_windows[63],
    update_CS(DM, CS, NewCS, ValidMove, NewPlayer),
    dialog(NewCS, FS, [(NewPlayer, ValidMove, NewLevels)|DM]).
  }

```

This macro is invoked in definition 76.

In `move_input/5` the next move is done. This clause returns the performed move and the current levels of the dialog. Clause (`valid_move/6`) checks the validity of the latest move. Note that the validity of a forced move is checked although it has to be always valid. In case the move turns out to be invalid, the player is asked to enter another move instead of the latest one. When a valid move has been obtained, clause `update_CS/5` updates the commitment store. Finally `dialog/3` is started again for the rest of the dialog.

## 4 The moves

Perform the next move. First check whether a forced move has to be done, and perform this move automatically. If that is not the case, check whether a special rule applies. If no special rule applies, perform the case as a default.

```
get the next move[6] ≡
  {the forced moves[7]
   general move input[17]
  }
```

This macro is invoked in definition 76.

### 4.1 The forced moves

*Rule 16b.* If a player accepts the sentence `il_claim(S)`, then he is forced to withdraw `S`.

```
the forced moves[7] + ≡
  {move_input( CS
    , [(P, (accept, il_claim(S)), (L, _))|_]
    , P
    , (L, il_claim(S))
    , (withdraw, S)
    ):-
    otherplayer(P, Op),
    only_inCS_other(CS, Op, S),
    teller,
    write_status([(P, (accept, S), (L, _))|_], P, (L, S)).
  }
```

This macro is defined in definitions 7, 8, 9, 10, 11, 12, 13, 14, 15, and 16.

This macro is invoked in definition 6.

*Rule 7a.* A player is forced to accept that a rule is applicable, if he is committed to the statement that this rule is valid, and he is also committed to the statement that the precondition of this rule is satisfied.

First, deal with the situation that `P` accepted the precondition of the rule in the last move, whereas he accepted the rule itself previously.

```
the forced moves[8] + ≡
  {move_input( CS
    , [(P, (accept, A), (L, _))|_]
    , P
    , (L, A)
    , ( accept, reason( applicable(rule(A, B))
                      , applies(rule(A, B))
                    )
    )
    ) :-
    inCS((P, valid(rule(A, B))), CS),
```

```

    only_inCS_other( CS, P
                    , reason( applicable(rule(A, B))
                              , applies(rule(A, B)))
                    ),
    teller,
    write_status([(P, (accept, A), (L, _))|_], P, (L, A)).
}

```

This macro is defined in definitions 7, 8, 9, 10, 11, 12, 13, 14, 15, and 16.  
This macro is invoked in definition 6.

Next, deal with the situation that the validity of the rule was accepted in the latest move, and the precondition was recognized earlier.

```

the forced moves[9] + ≡
{move_input( CS
            , [(P, (accept, valid(rule(A, B))), (L, _))|_]
            , P
            , (L, valid(rule(A, B)))
            , (accept, reason(applicable(rule(A, B)), applies(rule(A, B))))
            ):-
    inCS((P, A), CS),
    only_inCS_other(CS, P, reason(applicable(rule(A, B)), applies(rule(A,
B))))),
    teller,
    write_status( [(P, (accept, valid(rule(A, B))), (L, _))|_]
                , P, (L, valid(rule(A, B)))
                ).
}

```

This macro is defined in definitions 7, 8, 9, 10, 11, 12, 13, 14, 15, and 16.  
This macro is invoked in definition 6.

Finally, if a player claims the reason that a rule is applicable (?) and his opponent is committed to the the preposition of the rule as well as to the validity to the rule, the opponent is forced to accept the reason that the rule is applicable.

```

the forced moves[10] + ≡
{move_input( CS
            , [( P
                , (claim, reason(applicable(rule(A, B)),
applies(rule(A,B))))
                , (L, _)
                )|_
            ]
            , Op
            , ( L
                , reason(applicable(rule(A, B)), applies(rule(A, B)))
                )
            , (accept, reason(applicable(rule(A, B)), applies(rule(A, B))))
            ):-
}

```

```

otherplayer(P, Op),
inCS((Op, A), CS),
inCS((Op, valid(rule(A, B))), CS),
teller,
write_status( [(P, (claim, reason(applicable(rule(A, B))
                        , applies(rule(A, B))))), (L, _) | _
              ]
              , Op
              , (L, reason( applicable(rule(A, B))
                            , applies(rule(A, B))
                            )
              )
              ).
}

```

This macro is defined in definitions 7, 8, 9, 10, 11, 12, 13, 14, 15, and 16.  
This macro is invoked in definition 6.

*Rule 8 a* Suppose a player is committed to the statement that a rule applies, but not committed on the reason based on that rule. If his opponent has claimed the reason based on that rule, the player is forced to accept it.

*the forced moves*[11] +  $\equiv$

```

{move_input( CS
            , [(P, (accept, applies(rule(A, B))), (L, _) | _]
            , P
            , (L, applies(rule(A, B)))
            , (accept, reason(A, B))
            ):-
only_inCS_other(CS, P, reason(A, B)),
teller,
write_status([(P, (accept, applies(rule(A, B))), (L, _) | _],
            P, (L, applies(rule(A, B)))).
}

```

This macro is defined in definitions 7, 8, 9, 10, 11, 12, 13, 14, 15, and 16.  
This macro is invoked in definition 6.

*Rule 8 b* Suppose a player is committed to the statement that a rule is excluded. If he previously claimed this still open sentence, then he is forced to withdraw that claim. Furthermore, he is not allowed to claim that that the rule applies.

*the forced moves*[12] +  $\equiv$

```

{move_input( CS
            , [(P, (accept, excluded(rule(A, B))), (L, _) | _]
            , P
            , (L, excluded(rule(A, B)))
            , (withdraw, applies(rule(A, B)))
            ):-

```

```

    otherplayer(P, Op),
    only_inCS_other(CS, Op, applies(rule(A, B))),
    teller,
    write_status([(P, (accept, applies(rule(A, B))), (L, _))|_], P,
                (L, applies(rule(A, B)))).
}

```

This macro is defined in definitions 7, 8, 9, 10, 11, 12, 13, 14, 15, and 16.  
This macro is invoked in definition 6.

*Rule 12 a* If a player accepts that arguments pro a statement  $S$  outweigh arguments contra  $S$ , but he is committed to  $\neg S$ , then he is forced to withdraw  $\neg S$ .

```

the forced moves[13] + ≡
{move_input( CS
    , [(P, (accept, outweighs(Pro, Con, S)), (L, _))|_]
    , P
    , (L, outweighs(Pro, Con, S))
    , (withdraw, not(S))
    ):-
inCS((P, not(S)), CS),
teller,
write_status( [(P, (accept, outweighs(Pro, Con, S)), (L, _))|_]
    , P
    , (L, outweighs(Pro, Con, S))
    )
.
}

```

This macro is defined in definitions 7, 8, 9, 10, 11, 12, 13, 14, 15, and 16.  
This macro is invoked in definition 6.

If a player has accepted that the pro arguments outweigh the contra arguments with respect to statement  $S$ , and, as a consequence, has withdrawn  $\neg S$ , he is ready to accept  $S$ .

```

the forced moves[14] + ≡
{move_input( CS
    , [(P, (withdraw, not(S)), (L, _))|_]
    , P
    , (L, not(S))
    , (accept, S)
    ):-
inCS((P, outweighs(Pro, Con, S)), CS),
teller,
write_status( [(P, (withdraw, not(S)), (L, _))|_]
    , P
    , (L, outweighs(Pro, Con, S))
    )
.
}

```

This macro is defined in definitions 7, 8, 9, 10, 11, 12, 13, 14, 15, and 16.  
This macro is invoked in definition 6.

If a player accepts that the arguments pro statement  $S$  outweigh the statements contra  $S$ , then he is forced to accept the statement  $S$  itself.

```

the forced moves[15] + ≡
  {move_input( _
    , [(P, (accept, outweighs(Pro, Con, S)), (L, _))|_]
    , P
    , (L, outweighs(Pro, Con, S))
    , (accept, S)
    ):-
  teller,
  write_status( [(P, (accept, outweighs(Pro, Con, S)), (L,_))|_]
    , P
    , (L, outweighs(Pro, Con, S))
    )
  .
}

```

This macro is defined in definitions 7, 8, 9, 10, 11, 12, 13, 14, 15, and 16.  
This macro is invoked in definition 6.

*Rule 14 c1* If a player accepts that a statement  $S$  is not valid because of a reason  $R$ , but he was committed to the statement that arguments pro  $S$  outweigh the arguments contra  $S$ , then he is forced to withdraw the latter commitment.

```

the forced moves[16] + ≡
  {move_input(CS
    , [(P, (accept, reason(R, not(S))), (L, _))|_]
    , P
    , (L, reason(R, not(S)))
    , (withdraw, outweighs(Pro, Con, S))
    ):-
  otherplayer(P, Op),
  only_inCS_other(CS, Op, outweighs(Pro, Con, S)),
  teller,
  write_status([(P, (accept, reason(R, not(S))), (L, _))|_], P,
    (L, reason(R, not(S)))).
  }

```

This macro is defined in definitions 7, 8, 9, 10, 11, 12, 13, 14, 15, and 16.  
This macro is invoked in definition 6.

## 4.2 General move input (voluntary, non-forced moves)

Determine what the level of the next move is, and which player's turn it is (function `next_move`). Write the status and let the player perform the move.

```

general move input[17] + ≡

```

```

{move_input(CS, DM, NewPlayer, NewLevels, Move):-
    teller,
    next_move(DM, CS, NewPlayer, NewLevels),
    write_status(DM, NewPlayer, NewLevels),
    read_move(NewPlayer, Move).
}

```

This macro is defined in definitions 17, 18, 19, 20, 29, and 30.  
This macro is invoked in definition 6.

The new move, and who the new player is, depends on the previous move and previous player. The `next_move` statement analyses `DialogMoves` and obtains values for `NewPlayer` and `Newlevels` (in other words, `DialogMoves` and `commitmentStore` are input variables, and `NewPlayer` and `Newlevels` are output variables).

```
next_move(DialogMoves, commitmentStore, NewPlayer, Newlevels)
```

#### 4.2.1 Response to a claim

In case a player claims a statement, let his opponent respond.

```

general move input[18] + ≡
{next_move( [(P, (claim, B), (L, _))|_]
    , -
    , NewPlayer
    , (L, B)
    ) :-
    otherplayer(P, NewPlayer).
}

```

This macro is defined in definitions 17, 18, 19, 20, 29, and 30.  
This macro is invoked in definition 6.

#### 4.2.2 Response to a question

If player questions a statement, increase the level, and let his opponent respond to the question.

```

general move input[19] + ≡
{next_move( [(P, (question, _), (L, B))|_]
    , -
    , NewPlayer
    , (LL, B)
    ) :-
    otherplayer(P, NewPlayer),
    LL is L +1.
}

```

This macro is defined in definitions 17, 18, 19, 20, 29, and 30.  
This macro is invoked in definition 6.

### 4.2.3 Response to accept and withdrawal

Several rules have to be checked in case a player accepted or withdrew a statement. Only if none of these rules are applicable, handle the case as a general case.

```
general move input[20] + ≡
  {check rules for acceptance or withdrawal[23]
   default handling of acceptance or withdrawal[21]
  }
```

This macro is defined in definitions 17, 18, 19, 20, 29, and 30.  
This macro is invoked in definition 6.

*general accept and withdrawal* Generally, if a player  $P$  accepts a statement  $S$ , the level of the discussion returns to the level where  $S$  has been claimed, and the other player is allowed to perform a move. This behaviour is prescribed by rule 5c2.

```
default handling of acceptance or withdrawal[21] + ≡
  {next_move( [(P, (accept, S), _) | Rest]
              , -
              , NewPlayer
              , (L, B)
              ) :-
    find_move(Rest, S, (L, B)),
    otherplayer(P, NewPlayer).
  }
```

This macro is defined in definitions 21 and 22.  
This macro is invoked in definition 20.

Generally, if a player withdraws a statement  $B$ , the level returns to the level where  $B$  was claimed, and the player is allowed to perform another move. This behaviour is prescribed by rule 5b2.

```
default handling of acceptance or withdrawal[22] + ≡
  {next_move( [(P, (withdraw, B), _) | Rest]
              , -
              , P
              , NewLevels
              ) :-
    find_move(Rest, B, NewLevels).
  }
```

This macro is defined in definitions 21 and 22.  
This macro is invoked in definition 20.

*Rule 5 a* If player  $P$  withdraws a statement  $B$ , and his opponent is committed to  $\neg B$ ,  $P$  is on turn, and the level goes to the value where  $B$  (actually  $\neg B$ ) has been claimed.

```
check rules for acceptance or withdrawal[23] + ≡
  {next_move( [(P, (withdraw, B), _) | Rest]
              , CS
```

```

    , P
    , (L, not(B))
  ) :-
  only_inCS_other(CS, P, not(B)),
  find_move(Rest, not(B), (L, _)).
}

```

This macro is defined in definitions 23, 24, 25, 26, 27, and 28.  
This macro is invoked in definition 20.

*Rule 5 c1* If a player  $P$  accepts the negation of a statement  $S$ , then the level of the discussion must return to the level where  $S$  has been claimed, and  $P$  must perform another move.

```

check rules for acceptance or withdrawal[24] + ≡
{ 'next_move( [(P, (accept, not(S)), _) | Rest]
    , -
    , P
    , (L, B)
  ) :-
  find_move(Rest, S, (L, B)).
}

```

This macro is defined in definitions 23, 24, 25, 26, 27, and 28.  
This macro is invoked in definition 20.

*Rule 5 b1* If a player withdraws the negation of a statement  $S$ , then the other player, who is supposed to having claimed  $S$ , is allowed to continue on the level where  $S$  had been claimed.

```

check rules for acceptance or withdrawal[25] + ≡
{ next_move( [(P, (withdraw, not(S)), _) | Rest]
    , -
    , NewPlayer
    , (L, B)
  ) :-
  find_move(Rest, S, (L, B)),
  otherplayer(P, NewPlayer).
}

```

This macro is defined in definitions 23, 24, 25, 26, 27, and 28.  
This macro is invoked in definition 20.

*Rule 16 c* This rule handles acceptance or withdrawal of  $il\_claim(S)$ . The player who claimed  $il\_claim(S)$  is allowed to perform the next move.

```

check rules for acceptance or withdrawal[26] + ≡
{ next_move( [(P, (withdraw, il_claim(S)), _) | Rest]
    , CS
    , P
    , (L, S)
  ) :-

```

```

    not_inCS( (_, not(il_claim(S))), CS),
    find_move(Rest, S, (L, _)).

next_move( [(P, (withdraw, not(il_claim(S))), _) | Rest]
    , CS
    , Op
    , (L, S)
    ) :-
    not_inCS( (_, il_claim(S)), CS),
    otherplayer(P, Op),
    find_move(Rest, S, (L, _)).

next_move( [(P, (accept, not(il_claim(S))), _) | Rest]
    , -
    , P
    , (L, S)
    ) :-
    find_move(Rest, S, (L, _)).
}

```

This macro is defined in definitions 23, 24, 25, 26, 27, and 28.  
This macro is invoked in definition 20.

*Rule 14 a* If a player withdraws a statement that a pro argument outweighs a con arguments with respect to a statement  $S$ , the level goes back to the level where the con argument was claimed.

```

check rules for acceptance or withdrawal[27] + ≡
{next_move( [(P, (withdraw, outweighs(_, _, S)), _) | Rest]
    , CS
    , P
    , (L, reason(R, not(S)))
    ) :-
    only_inCS_other(P, reason(R, not(S)), CS),
    find_move(Rest, reason(R, not(S)), (L, _)).
}

```

This macro is defined in definitions 23, 24, 25, 26, 27, and 28.  
This macro is invoked in definition 20.

*Rule 14 b* If a player accepts that a statement  $R$  is an argument pro  $\neg S$ , then find out whether there was an outweigh claim with respect to  $S$ . If so, find the level where the outweigh has been claimed, and check whether the outweigh is still in the commitment store. Finally, find the statement that the outweigh move was about ( $B$ ). The latter procedure seems redundant. The outweigh statement should be claimed in a move about  $S$  itself. The player who performed the accept move has to perform the next move.

```

check rules for acceptance or withdrawal[28] + ≡
{next_move( [(P, (accept, reason(R, not(S))), _) | Rest]

```

```

    , CS
    , P
    , (L, B)
  ) :-
  find_move(Rest, reason(R, not(S)), (L, outweighs(_, _, S))),
  inCS((_, outweighs(_, _, S)), CS),
  find_move(Rest, outweighs(_, _, S), (_, B)).

next_move( [(P, (withdraw, reason(R, not(S))), _) | Rest]
  , CS
  , NewPlayer
  , (L, B)
  ) :-
  find_move(Rest, reason(R, not(S)), (L, outweighs(_, _, S))),
  inCS((_, outweighs(_, _, S)), CS),
  find_move(Rest, outweighs(_, _, S), (_, B)),
  otherplayer(P, NewPlayer).
}

```

This macro is defined in definitions 23, 24, 25, 26, 27, and 28.  
This macro is invoked in definition 20.

*Rule 14 b general move input[29] + ≡*

```

{next_move([(P, (withdraw, reason(_, not(S))), _) | Rest], CS, P, (L, B)) :-
  inCS((_, outweighs(_, _, S)), CS),
  find_move(Rest, outweighs(_, _, S), (L, B)).
}

```

This macro is defined in definitions 17, 18, 19, 20, 29, and 30.  
This macro is invoked in definition 6.

#### 4.2.4 Find the level of a claim

Check at which level a statement has been claimed. The function `find_move` searches a list of dialog moves to find a move in which a statement  $S$  has been claimed. If it has been found, the term  $L$  contains on return the dialog level in which the claim took place.

*general move input[30] + ≡*

```

{find_move([( _ , (claim, S), (L, B)) | _], S, (L, B)).
  find_move([_ | Rest], S, NL) :-
    find_move(Rest, S, NL).
}

```

This macro is defined in definitions 17, 18, 19, 20, 29, and 30.  
This macro is invoked in definition 6.

### 4.3 Check the validity of moves

The general form of claus that cecks the validity of a move is:

```

valid_move(NewLevels, NewPlayer, CommitmentStore, DialogMoves, ValidMoves, ValidMove, Move)

```

It is always valid to quit. *check validity of moves*[31] + ≡

```
{valid_move(_, _, _, _, (quit, _), (quit, _)).
}
```

This macro is defined in definitions 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, and 47.  
This macro is invoked in definition 76.

Valid moves in case of accept:

*check validity of moves*[32] + ≡

```
{valid_move(_, NP, CS, _, (accept, S), (accept, S)) :-
    only_inCS_other(CS, NP, S),
    not_inCS(_, not(S)), CS).
/* valid moves in case of withdraw*/
valid_move(_, NP, CS, _, (withdraw, S), (withdraw, S)) :-
    otherplayer(NP, P),
    only_inCS_other(CS, P, S).
}
```

This macro is defined in definitions 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, and 47.  
This macro is invoked in definition 76.

Valid moves in case of question:

*check validity of moves*[33] + ≡

```
{valid_move(_, P, CS, [(P, (withdraw, S), _)|_],
    (question, not(S)), (question, not(S))) :-
    otherplayer(P, Op),
    inCS((Op, not(S)), CS).
}
```

This macro is defined in definitions 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, and 47.  
This macro is invoked in definition 76.

*check validity of moves*[34] + ≡

```
{valid_move(NL, NP, CS, DM, ValidMove, (question, outweighs([_], [],
_))):-
    read_again(NP, Move),
    valid_move(NL, NP, CS, DM, ValidMove, Move).
}
```

This macro is defined in definitions 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, and 47.  
This macro is invoked in definition 76.

*check validity of moves*[35] + ≡

```
{valid_move(NL, NP, CS, [(P, (question, S), L)|Rest], ValidMove,
(question, S)):-
    read_again(NP, Move),
    valid_move(NL, NP, CS, [(P, (question, S), L)|Rest], ValidMove,
Move).
}
```

This macro is defined in definitions 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, and 47.  
This macro is invoked in definition 76.

*check validity of moves*[36] + ≡

```

    {valid_move(NL, NP, CS, _, ValidMove,
                (question, reason(applicable(rule(A, B))), applies(rule(A,
B))))):-
        inCS((NP, A), CS),
        inCS((NP, valid(rule(A, B))), CS),
        read_again(NP, Move),
        valid_move(NL, NP, CS, [(_, (question, _), _)|_], ValidMove,
Move).
    }

```

This macro is defined in definitions 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, and 47.  
This macro is invoked in definition 76.

```

check validity of moves[37] + ≡
    {valid_move(_, _, _, [(_, (claim, S), _)|_], (question, S), (question,
S))}.

```

```

    valid_move(_, _, CS, [(_, (withdraw, il_claim(S)), _)|_],
                    (question, S), (question, S)):-
        not_inCS((_, not(il_claim(S))), CS).
    valid_move(_, _, CS, [(_, (withdraw, not(il_claim(S))), _)|_],
                    (question, S), (question, S)):-
        not_inCS((_, il_claim(S)), CS).
    valid_move(_, _, _, [(_, (accept, not(il_claim(S))), _)|_],
                    (question, S), (question, S)).

    valid_move(_, NP, CS, [(_, (withdraw, reason(_, not(S))), _)|_],
                    (question, outweighs(_, _, S)), (question, outweighs(_,
_, S))):-
        otherplayer(NP, P),
        only_inCS_other(CS, P, outweighs(_, _, S)).
    }

```

This macro is defined in definitions 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, and 47.  
This macro is invoked in definition 76.

Valid moves in case of claim:

```

check validity of moves[38] + ≡
    {valid_move(NL, NP, CS, DM, ValidMove, (claim, S)) :-
        previous_withdraw(NP, S, DM),
        read_again(NP, Move),
        valid_move(NL, NP, CS, DM, ValidMove, Move).
    }

```

This macro is defined in definitions 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, and 47.  
This macro is invoked in definition 76.

```

check validity of moves[39] + ≡
    {valid_move(NL, NP, CS, [(P, (withdraw, S), Levels)|Rest],
                ValidMove, (claim, _)) :-
        otherplayer(P, Op),

```

```

        inCS((Op, not(S)), CS),
        read_again(NP, Move),
        valid_move(NL, NP, CS, [(P, (withdraw, S), Levels)|Rest],
ValidMove, Move).
    }

```

This macro is defined in definitions 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, and 47.  
This macro is invoked in definition 76.

```

check validity of moves[40] + ≡
    {valid_move(NL, NP, CS, [(P, (withdraw, il_claim(S)), Levels)|Rest],
        ValidMove, (claim, _)) :-
        read_again(NP, Move),
        valid_move(NL, NP, CS, [(P, (withdraw, il_claim(S)),
Levels)|Rest],
            ValidMove, Move).
        valid_move(NL, NP, CS, [(P, (withdraw, not(il_claim(S))), Levels)|Rest],
            ValidMove, (claim, _)) :-
            read_again(NP, Move),
            valid_move(NL, NP, CS, [(P, (withdraw, il_claim(S)),
Levels)|Rest],
                ValidMove, Move).
            valid_move(NL, NP, CS, [(P, (accept, not(il_claim(S))), Levels)|Rest],
                ValidMove, (claim, _)) :-
                read_again(NP, Move),
                valid_move(NL, NP, CS, [(P, (withdraw, il_claim(S)),
Levels)|Rest],
                    ValidMove, Move).
            }

```

This macro is defined in definitions 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, and 47.  
This macro is invoked in definition 76.

```

check validity of moves[41] + ≡
    {valid_move(_, NP, CS, _, (claim, applies(rule(A, B))),
        (claim, applies(rule(A, B)))) :-
        not_inCS((_, applies(rule(A, B))), CS),
        not_inCS((NP, excluded(rule(A, B))), CS).
        valid_move(NL, NP, CS, DM, ValidMove, (claim, applies(rule(_, _)))) :-
        read_again(NP, Move),
        valid_move(NL, NP, CS, DM, ValidMove, Move).
    }

```

This macro is defined in definitions 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, and 47.  
This macro is invoked in definition 76.

```

check validity of moves[42] + ≡
    {valid_move((_, applies(rule(A, B))), NP, CS, _,
        (claim, reason(applicable(rule(A, B)), applies(rule(A,
B))))),
        (claim, reason(applicable(rule(A, B)), applies(rule(A,
B)))) :-

```

```

        not_inCS(('_', reason(applicable(rule(A, B))), applies(rule(A,
B))))), CS),
        not_inCS((NP, not(A)), CS),
        not_inCS((NP, not(valid(rule(A, B))))), CS).
    valid_move(NL, NP, CS, DM, ValidMove,
                (claim, reason(applicable(rule(A, B))), applies(rule(A,
B)))) :-
        read_again(NP, Move),
        valid_move(NL, NP, CS, DM, ValidMove, Move).
    }

```

This macro is defined in definitions 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, and 47.  
This macro is invoked in definition 76.

```

check validity of moves[43] + ≡
    {valid_move(('_', _, _, [(_, (claim, B), _)|_], (claim, il_claim(B)),
                (claim, il_claim(B))) :-
        B \= il_claim(_).
    valid_move(NL, NP, CS, DM, ValidMove, (claim, il_claim(_))) :-
        read_again(NP, Move),
        valid_move(NL, NP, CS, DM, ValidMove, Move).
    }

```

This macro is defined in definitions 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, and 47.  
This macro is invoked in definition 76.

```

check validity of moves[44] + ≡
    {valid_move((_, B), _, CS, _, (claim, reason(A, B)),
                (claim, reason(A, B))):-
        not_inCS(('_', reason(A, B)), CS).
    valid_move((_, B), _, CS, _, (claim, reason(A, not(B))),
                (claim, reason(A, not(B)))) :-
        not_inCS(('_', reason(A, not(B))), CS).
    valid_move(('_', _, CS, [(_, (claim, outweighs(_, _, B)), _)|_],
                (claim, reason(A, not(B))), (claim, reason(A, not(B))))
:-
        not_inCS(('_', reason(A, not(B))), CS).
    valid_move(NL, NP, CS, DM, ValidMove, (claim, reason(_, _))) :-
        read_again(NP, Move),
        valid_move(NL, NP, CS, DM, ValidMove, Move).
    valid_move((_, B), _, CS, _, (claim, outweighs(Proset, Conset, B)),
                (claim, outweighs(Proset, Conset, B))) :-
        not_inCS(('_', outweighs(Proset, Conset, B)), CS),
        check_reasonsets(CS, Proset, Conset, B).
    }

```

This macro is defined in definitions 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, and 47.  
This macro is invoked in definition 76.

```

check validity of moves[45] + ≡
    {valid_move(('_', _, _, [(_, (claim, S), _)|_], (claim, not(S)), (claim,
not(S))):-

```

```

    S \= not(_).
valid_move(_, _, CS, [(_, (question, _), _)|_], (claim, S), (claim, S)):-
    not_inCS(_, S), CS),
    not_inCS(_, not(S)), CS).
valid_move(_, _, CS, [(_, (withdraw, _), _)|_], (claim, S), (claim, S)):-
    not_inCS(_, S), CS),
    not_inCS(_, not(S)), CS).
valid_move(_, _, CS, [(_, (accept, _), _)|_], (claim, S), (claim, S)):-
    not_inCS(_, S), CS),
    not_inCS(_, not(S)), CS).
valid_move(NL, NP, CS, DM, ValidMove, _) :-
    read_again(NP, Move),
    valid_move(NL, NP, CS, DM, ValidMove, Move).
}

```

This macro is defined in definitions 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, and 47.  
This macro is invoked in definition 76.

A non-empty proset is not allowed. Check it.

```

check validity of moves[46] + ≡
{check_reasonsets(_, [], _, _):- fail.
 check_reasonsets(CS, Proset, Conset, B) :-
    peel(CS, Proset, Conset, B, _, _).
}

```

This macro is defined in definitions 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, and 47.  
This macro is invoked in definition 76.

```

check validity of moves[47] + ≡
{peel([(bert, reason(R, B))|Rest], Proset, Conset, B, Checkpro, Checkcon)
 :-
    peel(Rest, Proset, Conset, B, [R|Checkpro], Checkcon).
 peel([(bert, reason(R, not(B))|Rest], Proset, Conset, B, Checkpro,
 Checkcon) :-
    peel(Rest, Proset, Conset, B, Checkpro, [R|Checkcon]).
 peel([_|Rest], Proset, Conset, B, Checkpro, Checkcon) :-
    peel(Rest, Proset, Conset, B, Checkpro, Checkcon).
 peel([], Proset, Conset, _, Proset, Conset).
}

```

This macro is defined in definitions 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, and 47.  
This macro is invoked in definition 76.

## 5 The commitment store

### 5.1 Update the commitment store

There are two update clauses:

```

update_CS(DialogMoves, CS, NewCS, Move, NewPlayer)
bigupdate_CS(DialogMoves, CommitmentStore, NewCS, ValidMove,
NewPlayer)

```

```

update the commitment store[48] + ≡
{update_CS(_, CS, [(NP, S)|CS], (claim, S), NP):-
    write_CS([(NP, S)|CS]).
}

```

This macro is defined in definitions 48, 49, 50, 51, and 74.  
This macro is invoked in definition 76.

Delete elements:

```

update the commitment store[49] + ≡
{update_CS(DM, CS, NewCS, (accept, S), NP) :-
    bigupdate_CS(DM, CS, NewCS, (accept, S), NP),
    write_CS(NewCS).
update_CS(DM, CS, NewCS, (withdraw, S), NP) :-
    bigupdate_CS(DM, CS, NewCS, (withdraw, S), NP),
    write_CS(NewCS).
update_CS(_, CS, CS, _, _):-
    write_CS(CS).
}

```

This macro is defined in definitions 48, 49, 50, 51, and 74.  
This macro is invoked in definition 76.

```

update the commitment store[50] + ≡
{bigupdate_CS([(_, (claim, S), _)|_], CS, [(NP, S)|CS], (accept, S), NP).
bigupdate_CS([X|Rest], CS, NewCS, (accept, S), NP) :-
    bigupdate_CS(Rest, CS, NextCS, (accept, S), NP),
    realupdate_CS([X|Rest], NextCS, NewCS, (accept, S), NP).
bigupdate_CS([(_, (claim, S), _)|_], CS, NewCS, (withdraw, S), NP) :-
    del((NP, S), CS, NewCS).
bigupdate_CS([X|Rest], CS, NewCS, (withdraw, S), NP) :-
    bigupdate_CS(Rest, CS, NextCS, (withdraw, S), NP),
    realupdate_CS([X|Rest], NextCS, NewCS, (withdraw, S), NP).
}

```

This macro is defined in definitions 48, 49, 50, 51, and 74.  
This macro is invoked in definition 76.

The actual big update takes place in a `realupdate_CS`:

```

realupdate_CS(DialogMoves, CommitmentStore, NewCS, ValidMove, NewPlayer)

```

```

update the commitment store[51] + ≡
{realupdate_CS([(P, (claim, not(S)), _)|_], CS, CS, (withdraw, S), _) :-
    inCS((P, not(S)), CS).
realupdate_CS([(P, (claim, S), _)|_], CS, NewCS, _, _) :-
    inCS((P, S), CS),
    otherplayer(P, Op),
    not_inCS((Op, S), CS),
    del((_, S), CS, NewCS).
realupdate_CS(_, CS, CS, _, _).
}

```

This macro is defined in definitions 48, 49, 50, 51, and 74.  
This macro is invoked in definition 76.

### 5.1.1 Consult the commitment store

```
consult the commitment store[52] + ≡
  {only_inCS_other(CS, P, S) :-
    otherplayer(P, Op),
    inCS((Op, S), CS),
    not_inCS((P, S), CS).
  }
```

This macro is defined in definitions 52, 53, and 54.  
This macro is invoked in definition 76.

```
consult the commitment store[53] + ≡
  {inCS(_, []) :- fail.
    inCS(Element, [Element|_]).
    inCS(Element, [_|Rest]) :-
      inCS(Element, Rest).
  }
```

This macro is defined in definitions 52, 53, and 54.  
This macro is invoked in definition 76.

```
consult the commitment store[54] + ≡
  {not_inCS(_, []).
    not_inCS(Element, [OtherElement|Rest]) :-
      Element \= OtherElement,
      not_inCS(Element, Rest).
  }
```

This macro is defined in definitions 52, 53, and 54.  
This macro is invoked in definition 76.

## 6 Other elements of Dialaw

### 6.1 Change the players

```
change the players[55] ≡
  {otherplayer(bert, ernie).
    otherplayer(ernie, bert).
  }
```

This macro is invoked in definition 76.

### 6.2 previous withdrawal

```
previous withdrawal[56] ≡
  {/*previous_withdraw(Player, S, DM)*/
    previous_withdraw(_, _, []) :- fail.
    previous_withdraw(P, S, [(P, (withdraw, S), _)|_]).
    previous_withdraw(P, S, [_|Rest]) :-
      previous_withdraw(P, S, Rest).
  }
```

This macro is invoked in definition 76.

### 6.3 The move counter

```
initialize the move counter[57] ≡
  {retract(teller(_))}
```

This macro is invoked in definition 76.

```
set the move counter to unity[58] ≡
  {      assert(teller(1)), teller(U),}
```

This macro is invoked in definition 2.

```
update the move counter[59] ≡
  {teller :-
      retract(teller(Umin1)),
      U is Umin1 + 1,
      assert(teller(U)).}
```

This macro is invoked in definition 76.

### 6.4 Some code that I don't understand

```
delX[60] ≡
  {del(X, [X|Tail], Tail).
   del(X, [Y|Tail], [Y|Tail1]) :-
     del(X, Tail, Tail1).
  }
```

This macro is invoked in definition 76.

## 7 Input and output

The users have to pass moves to the program and to read information about the status of the dialog. Standard Prolog has little facilities to communicate with the user(s). Therefore we will use a Prolog extension for graphical user interfaces. As a consequence, much of the code in this chapter is not standard Prolog.

### 7.1 Low level input from/output to the terminal

Print a character string. The following code has been copied from [1]

```
low-level read/write[61] + ≡
  {printreeks([]).
   printreeks([K|S]) :- put(K), printreeks(S).
  }
```

This macro is defined in definitions 61.

This macro is invoked in definition 76.

## 7.2 Windows management and user interface

The status of the dialog is showed in a panel with four windows. At the bottom, two windows show the commitments of the two players, Bert at the left side, Ernie at the right side. On top of these two windows there are three windows on top of each other. The lowest of the three lists the acts of the two players (*dialog* window), the middle window shows the current move and the top window shows the program status.

```
create/delete windows[62] + ≡
  {create_windows :-
    new(@mainf, frame('Dialaw Window')),
    send(@mainf, append, new(@csb, view('Berts commitments',
size(25,10))))),
    send(new(@cse, view('Ernies commitments', size(25,10))), right, @csb),
    send(new(@dv, view('Dialog', size(@default,10))), above, @csb),
    send(new(@sv, view('status', size(@default,10))), above, @dv),
    send(@mainf, open).
  }
```

This macro is defined in definitions 62 and 64.

This macro is invoked in definition 76.

Clear windows.

```
clear the commitment windows[63] ≡
  { send(@cse, clear),
    send(@csb, clear)
  }
```

This macro is invoked in definition 5.

When the dialog has been finished, delete the window panel.

```
create/delete windows[64] + ≡
  {delete_windows :-
    send(@mainf, destroy).
  }
```

This macro is defined in definitions 62 and 64.

This macro is invoked in definition 76.

Write a string to one of the windows in the panel with the following macro The macro has two arguments: the name of the window and the string to be written. *write in screen-*

```
partition[65](◊2)M ≡
  { send(◊1, print, ◊2)
  }
```

This macro is invoked in definitions 67, 68, 68, 68, 68, 68, 68, and 68.

Write a string to a window, and append the string with a newline character: *write line in*

```
screen-partition[66](◊2)M ≡
  { send(◊1, print, ◊2),
    send(◊1, newline) }
```

This macro is invoked in definition 67.

Write the first status, before asking for the first statement. *write the first status*[67] ≡

```
{   write in screen-partition[65]('@sv','U'),
    write line in screen-partition[66]('@sv',
        '. On level 0, move 1 of the dialog game'), }
```

This macro is invoked in definition 2.

Update the windows after the first move:

```
update the screen after the initial move[68] ≡
{   write in screen-partition[65]('@dv','1. bert: '),
    write in screen-partition[65]('@csb','U'),
    write in screen-partition[65]('@csb','. '),
    write in screen-partition[65]('@csb','Sentence'),
    write in screen-partition[65]('@cse','U'),
    write in screen-partition[65]('@cse','. '),
    write in screen-partition[65]('@cse','[]')}
```

This macro is invoked in definition 2.

Write the new status on the screen after other moves:

```
write_status(DialogMoves, NewPlayer, NewLevels)
```

Update the information on the windows. *write the status*[69] ≡

```
{write_status([(_, (A, S), _)|_], P, (L, N)) :-
    teller(U),
    send(@sv, print,U),
    send(@sv, print, '. The level is: '),
    send(@sv, print, L),
    send(@sv, print, '; the move is about '),
    send(@sv, print, N),
    send(@sv, newline),
    send(@dv, clear),
    send(@dv, print, A),
    send(@dv, print, ', '),
    send(@dv, print, S),
    send(@dv, newline),
    send(@dv, print, U),
    send(@dv, print, '. '),
    send(@dv, print, P),
    write_levelmark(L).
}
```

This macro is invoked in definition 76.

Write the level mark in the dialog window. *write the level mark*[70] ≡

```
{write_levelmark(0) :-!.
write_levelmark(L) :-
    send(@dv, print, '>>'),
    LL is L-1,
```

```

        write_levelmark(LL).
    }

```

This macro is invoked in definition 76.

The user input occurs via pop-up windows. The first move is always a claim of Bert, so, we only have to read the contents of the claim. This code is stolen from the XPCE manual [4] (pp. 32–33).

```

ask Bert for Sentence[71] ≡
{
    new(D, dialog('Berts statement:')),
    send(D, width, 500),
    send(D, append,
        new(TI, text_item(statement, ''))),
        send(TI, geometry,@default,@default,450,@default ),
    send(D, append,
        button(ok, message(D, return,
            TI?selection))),
    send(D, append,
        button(cancel, message(D, return, @nil))),
    send(D, default_button, ok), % Ok: default button
    get(D, confirm, Answer),    % This blocks!
    send(D, destroy),
    Answer \== @nil,           % canceled
    Sentence = Answer
}

```

This macro is invoked in definition 2.

Read the next move of the player(`read_move(NewPlayer, Move)`).

```

read the next move[72] ≡
{read_move(P, (A, S)) :-
    new(@act, frame(P)),
    send(@act, append, new(D, dialog(P))),
    send(D, width, 500),
    send(D, append, new(TI, text_item(sentence, ''))),
    send(TI, geometry,@default,@default,450,@default ),
    send(D, append, button(accept, message(D, return, accept))),
    send(D, append, button(withdraw, message(D, return, withdraw))),
    send(D, append, button(question, message(D, return, question))),
    send(D, append, button(claim, message(D, return, claim))),
    get(D, confirm, A),      % This blocks!
    set_act_type(A),
    get(TI, selection, S),
    act_type(A),
    send(@act, destroy).

set_act_type(Value) :-
    retractall(act_type(_)),
    assert(act_type(Value)).

```

```

write_dialog([]).
write_dialog([K|T]):-
    send(@dv, print, K),
    write_dialog(T).

}

```

This macro is invoked in definition 76.

If a move was wrong, try again. Unfortunately, on entry the variables A and S are not filled it. Therefore, this clause cannot inform the user what the wrong move was. *re-read the next*

```

move[73] ≡
{read_again(P, (A, S)) :-
    new(@wrong, dialog('Wrong move')),
    send(@wrong, append, new(@wm, view)),
    send(@wm, newline),
    send(@wrong, append, button(ok, message(@wrong, destroy))),
    send(@wrong, open),
    read_move(P, (A, S)),
    send(@wm, destroy),
    send(@wrong, destroy).
}

```

This macro is invoked in definition 76.

re-read a move)read\_again)

Write the Commitment store to the screen: This part deviates from Lodder's original program [3]. Lodder used, apart from the `write_CS/1` clause, also `write_CS/4` clauses, in which he disassembled the commitment store into statements to which Bert is committed and statements to which Ernie is committed. Unfortunately they did not work for me, although I don't know why.

```

update the commitment store[74] + ≡
{write_CS([(bert, S)|Rest]) :-
    teller(U),
    send(@csb, newline),
    send(@csb, print, U),
    send(@csb, print, '. '),
    send(@csb, print, S),
    write_CS(Rest).

write_CS([(ernie, S)|Rest]) :-
    teller(U),
    send(@cse, newline),
    send(@cse, print, U),
    send(@cse, print, '. '),
    send(@cse, print, S),
}

```

```
write_CS(Rest).
```

```
write_CS([]).  
}
```

This macro is defined in definitions 48, 49, 50, 51, and 74.  
This macro is invoked in definition 76.

```
stop the program[75] ≡  
{  
    new(@stop, dialog('Stop')),  
    send(@stop, append, button(stop, message(@stop, return, doit))),  
    send(@stop, default_button, stop),  
    send(@stop, open),  
    get(@stop, confirm, _),  
    send(@mainf, destroy),  
    send(@stop, destroy)}  
}
```

This macro is invoked in definition 1.

## 8 Put everything in a Prolog file

```
Dialaw.pl[76] ≡  
{/*DiaLaw 2.0, 1998, Prolog 2 for Windows*/  
clean :-  
    initialize the move counter[57],  
    fail.  
clean.  
  
the main clause[1]  
the first move[2]  
the conditions under which dialog/3 ends[3]  
the main dialog clause[5]  
get the next move[6]  
write the status[69]  
read the next move[72]  
reread the next move[73]  
check validity of moves[31]  
update the commitment store[48]  
change the players[55]  
previous withdrawal[56]  
consult the commitment store[52]  
delX[60]  
create/delete windows[62]  
update the move counter[59]  
write the level mark[70]  
low-level read/write[61]  
}
```

This macro is attached to an output file.

## References

- [1] WF Clocksin and CS Mellish. *Prolog, beschrijving van de standaard*. Kluwer, Deventer, 1987. Vertaald. Oorspr. titel: "Programming in Prolog".
- [2] Donald E. Knuth. *Literate Programming*. Number 27 in CLSI lecture notes. Stanford, California, 1992. ISBN 0-937073-80-6.
- [3] A. Lodder. *Dialaw: onlegal justification and dialog games*. PhD thesis, University of Maastricht, 1998.
- [4] Jan Wielemaker and Anjo Anjewierden. *Programming in XPCE/Prolog*. SWI, University of Amsterdam, the Netherlands, Roetersstraat 15 1018 WB Amsterdam, the Netherlands, for xpce version 14.9.3 edition, 1997.

## 9 Indexes

## Scrap index

(, 27

change the players, 22

check validity of moves, 16–20

consult the commitment store, 22

create/delete windows, 24

delX, 23

Dialaw.pl, 28

find the level of a claim, 15

general move input, 11–15

initialize the move counter, 23

low-level read/write, 23

previous withdrawal, 22

read the next move, 27

set the move counter to unity, 23

stop the program, 28

the conditions under which `dialog/3` ends,  
5

the first move, 4

the forced moves, 6–10

the main clause, 4

the main dialog clause, 5

update the commitment store, 21, 28

update the move counter, 23

update the screen after the initial move, 25

write in screen-partition, 24

write line in screen-partition, 24

write the first status, 25

write the level mark, 26

write the status, 25

## Identifier index

(, 27

accept, 5, 6

Act, 3

bert, 3, 4

claim, 4, 11

clean, 4

CommitmentStore, 3

CS, 4-6

dialaw, 4

dialog, 4, 5

DialogMove, 3

Dialogmove, 3

DialogMoves, 3, 4

DM, 5

ernie, 3

find\_move, 13, 15

first\_move, 4

FirstSentence, 4

FS, 5

il\_claim, 6

L, 6

Level, 3

Move, 5

move\_input, 5, 6, 11

MoveAbout, 3

NewCS, 5

NewLevels, 5

NewPlayer, 5

next\_move, 10-15

not\_inCS, 5

Only\_inCS\_other, 6

Op, 6

otherplayer, 6

outweighs, 9, 10

P, 6

Player, 3

question, 11

quit, 5

read\_move, 11, 16-20, 27

S, 6

Sentence, 3, 4

teller, 6

update\_CS, 5

ValidMove, 5

withdraw, 5, 6, 13

write\_status, 6

## **Index**

arity, 2

commitment store, 3

first\_move, 4

move, 3

player1, 3

player2, 3