CardsWorkShop V1.0 Help Contents

Dedicated to the memory of Sophie D'Amboise
à la mémoire d'une complicité parfaite

CWS Menu Commands

File Menu
- **New**
- **Open...**
- **Save**
- **Save As...**
- **Exit**

Edit Menu
- **Undo**
- **Cut**
- **Copy**
- **Paste**
- **Clear**
- **Find...**
- **Replace...**
- **Next**

Game Menu
- **Compile**
- **Run**
- **Open List**
- **Open...**
- **Choose Deck...**
- **Options...**

Play Menu
- **Rules**
- **ReStart**
- **Enter Seed...**
- **Redraw**

Start
Undo

File|New

New opens a new Edit window with the default name <ANONYMOUS> and automatically makes the new Edit window active.

These anonymous files are used as a temporary edit buffer.

CWS prompts you to name an anonymous file when you save it.

File|Open...

The File|Open dialog box appears. It is where you open a file by typing the file name in the input box or using the list boxes to find and open the file.

File Name input box

The File Name input box is where you enter the name of the file to load, or the file-name mask to use as a filter for the Files list box.

Files list box

The Files list box lists the names of files in the current directory that match the file-name mask in the File Name input box, plus the parent directory and all subdirectories.

Directories list box

You view the contents of different directories by selecting a directory name in the Directories list box.

File|Save

The Save command saves the file in the active Edit window to disk.

If the file has a default name (<ANONYMOUS>), CWS opens the File Save As dialog box so you can rename the file and save it in a different directory or on a different drive.

File|Save As...

Save As opens up the File Save As dialog box, where you can save the file in the active Edit window under a different name, in a different directory, or on a different drive.

The File Save As dialog box is where you type in the new name in the File Name input box (you can include a drive and directory path) or use the Directories list to select a new path.

If you choose an existing file name, CWS asks if you want to overwrite the existing file.

The window containing this file is updated with the new name.

File|Exit

The Exit command exits CWS and removes it from memory.

If you've modified a source file without saving it, CWS prompts you to do so before exiting.

Also, you can press Alt+F4 to exit.

Edit|Undo or Undo

In an Edit Window :

The Undo command "undoes" the most recent edit or cursor movement.

Undo inserts any characters you deleted, deletes any characters you inserted, replaces any characters you overwrote, and moves your cursor back to a prior position.

If you undo a block operation, your file will appear as it was before you executed the block operation.

In a Game Window :

The Undo command "undoes" your most recent transaction on the playfield.

If you continue to press Undo, it continues to undo the changes you made during the current game.

Edit|Cut

The Cut command removes the selected text from your document and places the text in the Clipboard.

You can then choose Edit|Paste to paste the cut text into any other document (or somewhere else in the same document).

The text remains selected in the Clipboard so you can paste it as many times as you want.

Edit|Copy

The Copy command leaves the selected text intact but places an exact copy of it in the Clipboard.

To paste the copied text into any other document, choose Edit|Paste.

Edit|Paste

The Paste command inserts the selected text from the Clipboard into the current window at the cursor position.

Edit|Clear

The Clear command removes the selected text but does not put it into the Clipboard.

This means you can't paste the text as you could if you had chosen Cut or Copy.

Although you can't paste the cleared text, you can undo the Clear command with Undo.

Edit|Find...

You use the Find Text dialog box to specify the text you want to search for.

Search for input box

This input box is where you enter the search string. Choose OK to begin the search, or choose Cancel to forget it.

Case Sensitive

When the Case Sensitive option is on, CWS differentiates uppercase from lowercase when performing a search.

Case Sensitive Off is the default.

Edit|Replace...

The Replace Text dialog box is where you specify the text to search for and what to replace it with.

Search for input box

Enter the search string in the Text to Find input box and choose OK to begin the search, or choose Cancel to forget it.

Replace with input box

Enter the replacement string in the New Text input box.

Case Sensitive

When the Case Sensitive option is on, CWS differentiates uppercase from lowercase when performing a search.

Case Sensitive Off is the default.

All Occurences

Set All Occurences on if you want CWS to replace all occurrences of the search string found.

Prompt On Replace

When the Prompt On Replace option is on, CWS prompts you before replacing each time it finds the search string.

When Prompt On Replace is off, CWS automatically replaces the search string.

Edit|Next

The Next command repeats the last Find or Replace command.

The last settings made in the Find Text or Replace Text dialog box remain in effect when you choose Next.

Game|Compile

The Compile command compiles the file in the active edit window.

If an error occurs, the status bar displays the error and a token near the error is highlighted.

Game|Run

The Run command runs the last compiled program.

Game|Open List

This command opend a new Games Icons List Box

Game|Open

This dialog works like the File|Open dialog, except that the file choosen must be an executable file (*.cvc) and is executed after selection.

Game|Choose Deck...

This opens a dialog in which you can choose the default deck of card for all games. Only one deck at a time is active.

Use the scroll bar to choose the deck and then press the Ok button.

Game|Options...

This opens a dialog in which you can set options for the whole system.

Moving Shadow

Turns moving shadow accompanying game transaction on or off.

Include *.cdl

Indicate if source files (*.cdl) get icons in Games Icons List Box

Include *.cvc

Indicate if executable files (*.cvc) get icons in Games Icons List Box

Include *.csg

Indicate if saved player game files (.csg) get icons in Games Icons List Box*

Play|Rules

Puts you in inspecting mode. In this mode when you click on a stack with a mouse button, information (rules of the game) will be displayed (if the game programmer created some).

Play|ReStart

Lets you start the current game over again.

Play|Enter Seed...

Lets you see the current game random seed and lets you change it.

Entering the same seed twice will permit you to play the same game twice.

Play|ReDraw

Redraw the current window.

In Games Icons List Box this command lets you update the list of icons if you just compiled a new game for the first time.

In Game Window this is usefull when the fireworks go awry.

Start

Starts a new game in the current Game Window.

Introduction to CardsWorkShop V1.0

CardsWorkShop is a integrated editor/compiler/player allowing the quick design and play of solitary card games. The language used ressembled PASCAL and is kinda object oriented (with only one type of object). A good number of examples are included.

This is the first version so your feedback will be appreciated. Don't forget to take notice of the ToDo section. And of course don't forget to register.

Quick How-to-use

Playing :

Double click on the icon representing the game you want to play.

Compiling :

Load a *.cdl file from the file|open menu or by double-clicking on its icon in a Games Icons List Box.

Compile it by pressing the appropriate button in the Game Window.

ToDo

Level 6

Modify icons to represent the game it is connected to

Add the possibility to have stack overlap (to program such games as pyramid...)

Add a stack type spread, wich will spread all its card on its surface instead of stacking them one on another

Add a type of card EmptySpot, wich is a see-through card

Level 7

Add construction to language to control the status bar in the bottom of the window. For example writing a timer, score...

Add HiScore dialog for games which computes score

Add loosing special-effects...

Add possibility to Save/Load game in progress

Level 8

Add a #include pre-processor

Add a byte-code peephole optimiser

Add a byte-code debugger

Version 2.0 ShareWare

Copyright for CardsWorkShop

All the source games are copyrighted David Jean, 1993 except CHARLES.CDL wich is copyrighted Charles-E. Jean, 1993.

Registration

Why should I register?

To get CWS-Library-I, a compilation of 25 ready-to-run with sources solitary card games

To know about Version 2.0 and CWS-Library-II before everybody else

How much will that cost me?

17.95$ U.S, or 19.95$ cdn, p&h included.
oversea, add 3.00$ cdn for p&h.
no c.o.d. please.

Specify if you want a 3.5" or 5.25" disk.

You can ease yourself by printing the file order.frm.

Where

David Jean
1976 Le Montagnais, B-109
Sherbrooke, Québec, Canada
J1K 2X9

Playing compiled games

Glossary of solitaire terms
Games Icons List Box
Game Window

Glossary

Values	As in most card games, Ace, Two, Three and so on , including the picture cards, Jacks, Queen, King.
Suits	Consisting of Clubs, Diamonds, Hearts, Spades.
Colors	Of which there are two, Red and Black.
The Tableau	Consists of single cards, groups, piles, which have their own purposes and limitations , as described in eacg game.
Foundations	Are cards upon which others are built to form complete sequences, thus terminating the game. The Foundations may be part of the original Tableau, or they may be established during play, according to the individual game.
Sequences	
Ascending	Run from a low card, usually an Ace, on up to the high card, as A, 2, 3, 4, 5, 6, 7, 8, 9, 10, J, Q, K.
Descending	Run from a high card , usually a King, on down to the low card, as K, Q, J, 10, 9, 8, 7, 6, 5, 4, 3, 2, A.
Auxiliary Cards	Belong to The Tableau, which may be built upon Foundations, or may be used for forming temporary sequences, according to the rules.
Rows	Are cards dealt crosswise in The Tableau, either singly or overlapping, as specified.
Columns	Are cards dealt vertically in The Tableau, either singly or overlapping, as specified.
The Stock	Is a term applied to the remainder of the pack after The Tableau has been arranged.
The Reserve	Is a packet or group of cards that is laid aside or specially retained for building on Foundations.
Available Cards	Are any that are free for building on Foundations, or for transfer to auxiliary cards or columns.
Blocked Cards	Are those which must in some way be released to become available.
Waste Pile	Consists of cards that can not be used when dealt and therefore must be laid aside. Some games are lost when all the stock has gone into the Waste Pile. Others allow the Waste Pile to be used as a new Stock, as specified in the rules of individual games.

Games Icons List Box

The purpose of the Game Icons List Box is to ease the work when playing with files, a little like the File Manager. The surface is divided in three parts described below.

Path Box

In the upper left corner is a Path Box. It indicates the current path for all the icons in the Icons Box. When you quit CWS this informations is saved in CWS.INI.

Directory Box

In the upper right corner is a drop down list diplaying all the directories and drives you can choose to change the path in the Path Box.

Icon Box

The lower, bigger part of the window displays icons connected to source files and executable files. You can set what kind of icons are displayed here by using the Game| Options.. menu. To run a executable file or to open a source file, double-click on its icon.

Game Window

The Game Window is the actual place where you play soliatire. It is one big playfield covered with different stacks depending on which game you are playing.

Transactions on the playfield are made with the mouse by clicking on stacks and dragging cards. Some stacks will respond automatically when you click a button on them while others will let you drag cards elsewhere.

Usually by choosing the Play|Rules item in the menu you can inspect the different stack and get informations about how the work.

You can Undo transactions at any time and up to the beginning of the game.

Creating new Solitaire Games

Introduction to Creation

CWS is the smallest form of object-oriented language. Only one object is defined and you can customize instance of that object. The customization can be complemented by inheritance of methods from other instances.

I Think that this approach could be better classified as Actors.

Here is a list of important concept in CWS :

Stack
Transaction
System Predicates
Log
Playfield

System Predicates

There are three global system predicates that can be defined by the programmer : integrity? to check system Integrity, win? to check if player has won and loose? to check if he has lost.

integrity?

Is executed after each transaction to check the system integrity. If False is returned, the last transaction is undone.

If absent, the system doesn't check for integrity. It is equivalent to always returning true.

This can also be used for special operations like turning some new cards side up.

```
predicate integrity? is
   begin
   with it do
      if IsSideUp?(it[it!]) then Turn it[it!] side up
   for A1, A2, A3, A4;
   return True;
   end;
```

win?

Is executed after each transaction to check if the game is won. If True is returned, the game is over and the player is told of his succes.

If absent, the game will never end with succes. It is equivalent to always returning false.

The win? predicate is always checked before the loose? predicate, so if they both return true the player win.

```
predicate win? is
   return (A1!=13) and (A2!=13) and (A3!=13) and (A4!=13);
```

loose?

Is executed after each transaction to check if the game is lost. If True is returned, the game is over and the player is told of his failure.

If absent (sometimes it is easier for the player to see he has lost than to program it), the game will never end with failure. It is equivalent to always returning false.

The win? predicate is always checked before the loose? predicate, so if they both return true the player win.

```
predicate loose? is
   return (D1!=0) and not MovePossibleOnTableau?;
```

Stack

A stack is a container for an ordered set of cards. A stack can contain 0, 1 or more cards (in CWS up to 204) and a bottom of pile drawing indicating the state of the pile. This drawing is usually a red cross, a green circle or a shaded card.

Each stack can respond to different messages sent to it by the playing environment : start of game, selection with the mouse, destination of a drop with the mouse or request for information.

The mouse itself is a stack with restriction (Cursor).

Transaction

A transaction is the transport of cards from one stack to another between the time the user press the mouse button and the time he releases it.

For the programmer this means :

First case :

A) A stack answer the mouse button selection message and send some cards on the mouse stack (Cursor)

B) The user moves the mouse without releasing the mouse button to another stack.

C) The user releases the button and the destination stack is informed of the drop. If this stack refuses the cards on the mouse stack (if he doesn't removed them all) then the transaction is cancelled, otherwise the transaction is accepted and completed.

Second case :

A) A stack answer the mouse button selection message and send some cards to other stacks on the playfield but none to the mouse stack (Cursor). The transaction is completed when the mouse button is released.

If the user, when dragging, releases the cards someplace that doesn't answer to the drop message, the transaction is cancelled.

Playfield

The playfield is divided in a serie of row and columns giving a big matrix. The size of this matrix is defined independently for each game in its header.

Each cell of the matrix is one square unit but will not necessarily be square physically on the screen. For example if you define the matrix to be 10 by 10, and the game is played on a 640 by 480, then each square of the matrix will be 64 by 48 pixels.

The space taken by each stack is described by defining a rectangular sub-matrix inside the screen matrix.

A restriction of the current version is that no two sub-matrices must overlap.

When the Game Window is resized the physical size of the matrix changes. Then the new size for the deck of card is computed and a new deck is generated.

The size of a card is computed like this : every stack is checked and we keep the minimum height and minimum width found. We take the physical size of the resulting minimum rectangle and try to fit the biggest possible card frame in it. The card frame is always at a ration of 2 horizontally for 3 vertically.

See also visual aspect

Visual Aspect

Every modified stack in the execution of a Method are redrawn at the end of the Method. This lets you do many operations on a stack (like turning cards, reversing some subsequence order, etc.) without worrying about the visual aspect.

Modified stacks are also redraw after the execution of the integrity? predicate or explicitly with the execution of the DRAW instruction.

Log

When a transaction is accepted it is added to a transactions log. At this point, the user can undo the last transaction by choosing the undo commands in the menu.

The user can undo every transaction up to the start of the game. The system logs transaction after the execution of every start method.

The programmer has nothing to do for all this, it is done automatically. It should be noted that only global variables are logged. So if you use working variables which don't need to be global, better make them local so to not overload the log.

CardsWorkShop language description

Types

CWS has only predefined simple types that are used in variables and functions definitions.

The following is a list of these types. Every definition explains also how to define constant of that type.

types ::= STACK | INDEX | CARD | INTEGER | BOOLEAN | STRING

INTEGER

Integers are the whole numbers you learned to count with (1, 5, -21, and 752, for example).

The allowed range is -32768 to 32767.

STACK

Variable of this type can be associated with any defines Stack. Any operation to a Stack can be applied to a Variable of type STACK.

The iteration variable in a with instruction is of type STACK.

INDEX

Indexes are used a indices to access cards on a stack. They are used inside the [...] in a stack.

Integers can be used too.

STRING

Strings are a combination on characters inside a couple of quote.

For example, 'This is a string'.

You can use before a special character inside a string :

\t	tab
\n	newline
\'	quote
\\	back-slash

BOOLEAN

Indicates variables which can have the value TRUE or FALSE.

CARD

A card acts like an integer, in fact :

0..12	is the range from Ace to King of Spade
13..25	Ace to King of Heart
16..38	Ace to King of Club
39..51	Ace to King of Diamond
52..103	is the range from Ace of Spade to King of Diamond but side down
104..155	is the range from Ace of Spade to King of Diamond side up but shaded
156	is the green circle
157	is the red cross

So simple integer arithmetic can be applied on a value of CARD type.

```
{****are c1 & c2 of different color}
predicate AlternateColor?(c1, c2 : Card) is
   return (((c1 / 13) + (c2 / 13)) mod 2) = 1;
```

Constant

gives an immutable value to an identifier. Right now only integer constant can be defined.

const_def ::= CONST const_elm (',' const_elm)* ';'
const_elm ::= *id* ':=' const_exp
const_exp ::= number

There exists many predefined constants in CWS :

```
const
   up = 0;
   down = 1;
   left = 2;
   right = 3;
   over = 4;

   shaded = 2;

   decksize = 52;

   emptycard = 156;
   crosscard = 157;

   spade = 0;
   heart = 13;
   club = 26;
   diamond = 39;

   ace = 0;
   deuce = 1;
   three = 2;
   four = 3;
   five = 4;
   six = 5;
   seven = 6;
   eight = 7;
   nine = 8;
   ten = 9;
   jack = 10;
   qheen = 11;
   king = 12;

   false = {falsity value};
   true =  {truth value};
```

STACK

A stack object is an aggregate of 10 fields : 5 constants, 2 variables and 3 methods including one that can be broke up in 2 or 4 parts.

stack_def ::= STACK *id* [FROM *id2*] IS stack_body END *id* ';'
stack_body ::= (const_Init | meth_Init)*
const_init ::= *id* ':=' const_exp ';'
meth_init ::= *id* [parms] (FROM *id2* | IS statement) ';'

Note that in stack_def, STACK *id* and END *id* must be the same *id*.

A stack object can inherit all or any fields from another one.
A stack definition can be split up in multiple parts.
A stack must be defined before it is referenced. You can use forward declaration to help you :

stack W1;

The five constants are :

- X
- Y
- W
- H
- Direction

The two variables are :

- !
- [...]

The three methods are :

- Start
- Select
- Help

Example of a Stack

STACK Inheritance

If FROM *id2* is present in stack_def then all fields defined to this point are duplicated in the new Stack.

If FROM *id2* is present in meth_init then this methods is duplicated from the *id2* Stack. If the method being defined can be broke up in multiple parts, then all lower methods are duplicated. See Select.

STACK Example

Here's an example of a Stack object :

```
stack W1 is
  X := 2;
  Y := 2;
  Direction := down;
  W := 2;
  H := 12;
  //****************************
  Start is
    begin
    Pull 6 From D1;
    Turn [1..3] Side Up;
    end;
  //****************************
  Select(Spos : Index) is
    begin
    Pull 1 To D2;
    end;
end W1;
```

A stack definition can be split up in multiple parts.

```
stack D1 is
  Select(Spos : index) is
    DoShade(Spos,King+Spade);
end D1;

stack C1 is ...

stack D1 is
  Start is
    Pull 1 from C1;
end D1;
```

D1 is the same object, methods and constant definitions are accumulated in D1. If methods or constant are redefined, the lower definition prevails.

!

! is represent the length (number of cards) on the Stack. It can be preceded by the stack it qualifies or else it qualifies the contextual object.

! is of type INDEX.

[..]

[...] is the array of cards in the Stack. It can be preceded by the stack it qualifies or else it qualifies the contextual object.

It is indexed by variable of type INDEX or INTEGER.

The card a position 0 is the empty pile drawing. It
it to EmptyCard, CrossCard or a shaded Card. By default it is EmptyCard (green circle).

X and Y constant

X and Y gives the upper left position of the Stack in the virtual matrix of the playfield defined in the game header.

W and H constant

W and H gives the width and height of the Stack in the virtual matrix of the playfield defined in the game header.

DIRECTION constant

Describes the way the card are stacked on one another. The possible values are : UP, DOWN, LEFT, RIGHT, OVER.

Contextual Object

Contrary to the old fashion way, when you call a procedure, function or predicate from an object method, this procedure is in the object context. That means that you can directly access the variables ([...] and !) of the calling Stack.

For example :

```
procedure DoShade(Spos : index; c1 : Card) is
  begin
  [Spos]:=c1;
  Turn [Spos] side shaded;
  Turn [!] Side down;
  end;

stack D1 is
  Select(Spos : index) is
    DoShade(Spos,King+Spade);
end D1;
```

In the DoShade procedure, [Spos] becomes implicitly D1[Spos] and [!] becomes D1[D1!].

START method

At the beginning of a game (after the user press Start) this method is called by the system. It is called for each stack in the order defined in the Order part of a program.

Example of a Stack

SELECT method

Select(Spos : Index)

This method is called to answer a message from the mouse. It means that the mouse button was pressed or released over the card a the Spos position in this Stack.

The Spos parameter must be present but its name can be customised to your taste.

This method can be subdivised in two disjoint set :

SelectRight(SPos : Index)
SelectLeft(SPos : Index)

Works like Select but differentiate between the left mouse button and the right mouse button.

SelectRightFrom(SPos : Index)
SelectRightTo(SPos : Index)
SelectLeftFrom(SPos : Index)
SelectLeftTo(SPos : Index)

Works like the SelectRight or SelectLeft but differentiate if a button is pressed or released.

For example if a Stack answer to Select, all messages (button right or left, pressed or released) are treated by the same code.

HELP method

When the Game Window is in inspecting mode (see Rules) and a mouse button is pressed over a Stack, the Help method of that stack, if any, is executed.

Usually this method will open a Text Box (using Clear), write text (using Write) and/or add button (using Wait).

ORDER

order_def ::= ORDER *id* (',' *id*)*

The body of a program end by an ORDER statement. It a list ordering the stacks. This order applies to initialisation (Start method) and general redrawing of stacks.

Generaly, the first stack is where you add the deck(s) or cards and shuffle them :

```
Add Ace+Spade .. King+Diamond;
Shuffle;
```

Then in the Start method of the other stacks you pull cards from that first stack (here C1) :

```
Pull 4 from C1;
```

Expression

expression ::= **expression ('+'|'-'|'*'|'/'|AND|OR|MOD|'<<'|'>>') expression | expression ('='|'<>'|'<='|'>=') expression | ('+'|'-'|NOT) expression | *id* ! | *id* '[' expression ['..' expression] ']' | *id* | ! | '[' expression ['..' expression] ']' | *integer* | '(' expression ')'**

Expressions are evaluated from left to right in short-circuit (meaning that when the value of a boolean expression is determined, evaluation stops).

This is the priority of the operators. Operator on a same line are at the same priority and are evaluated from left to right.

Hi

unary NOT + -

AND MOD * /

OR + -

= <> <= >=

Low

Header

header ::= **GAME *id* IS *integer* BY *integer***

The first *integer* describes the width of the playfield and the second *integer*, the height.

Program layout

program ::= **header** **';'** **(stack_def | const_def | var_def | pred_def | proc_def | func_def)*** **order_def** **'.'**

PROCEDURE

proc_def ::= **PROCEDURE *id* [parms] IS [var_def] statement ';'**
parms ::= **'(' var_list (';' var_list)* ')'**

A procedure is a program part that performs a specific action, often based on a set of parameters.

The procedure heading specifies the identifier for the procedure and the formal parameters (if any).

A procedure is activated by a procedure statement.

Since a procedure must be declared before being used and sometimes circular references would be useful, you can delacre a procedure before it is defined like this :

procedure D1(it : stack);

or

procedure D1;

PREDICATE

pred_def ::= **PREDICATE *id* [parms] IS [var_def] statement ';'**
parms ::= **'(' var_list (';' var_list)* ')'**

A predicate is a program part that computes a boolean value, often based on a set of parameters.

The predicate heading specifies the identifier for the predicate and the formal parameters (if any).

A predicate is activated in an expression.

Since a predicate must be declared before being used and sometimes circular references would be useful, you can delacre a predicate before it is defined like this :

predicate Empty?(it : stack);

or

predicate Empty?;

By convention a predicate name should end with a question mark.

There are three special predicates you can define :
- Win?
- Loose?
- Integrity?

FUNCTION

func_def ::= **FUNCTION *id* [parms] ':' types IS [var_def] statement ';'**
parms ::= **'(' var_list (';' var_list)* ')'**

A function is a program part that computes a value of type types, often based on a set of parameters.

The function heading specifies the identifier for the function and the formal parameters (if any).

A function is activated in an expression.

Since a function must be declared before being used and sometimes circular references would be useful, you can delacre a predicate before it is defined like this :

function Higher(it : stack): card;

or

function Higher;

VAR

var_def ::= **VAR (var_elm ';')***
var_elm ::= ***id*** **(',' *id*)* ':' types**

A variable (var) declaration associates an identifier and a type with a location in memory where values of that type can be stored.

Instruction

SELF variable

SELF is a variable that can be accessed inside any method or procedure, function or predicate called from a method. It is a variable of type STACK and referenced the current stack.

CURSOR variable

CURSOR is a Stack but with restriction. For one the Direction is always DOWN and cards can be on the CURSOR stack only while in a Transaction.

You don't have to declare it or to write methods for it.

Editor Window

Edit windows are where you type in and edit your CWS code. You can also do the following in an edit window:

- compile your programs
- run your programs
- read them from disk files
- save them to disk files

You can open as many edit windows as you want but each one are limited to around 32K of text.

To open an edit window, choose File|Open. You can open the same file in more than one window.

The button at the top of the window are shortcut for menu items of that name.

CLEAR

statement ::= CLEAR

Closes the HelpBox if it is open.

statement ::= CLEAR *string* [AT *integer* ',' *integer* IS *integer* BY *integer*]

Opens the HelpBox giving the title *string*. If the four *integer* are present, they specify a position (x, y) and a size (w, h) on the playfield matrix.

DRAW

statement ::= DRAW [*id*]

Forces the redrawing of Stack *id*.

WAIT

statement ::= WAIT *string id*

Adds a button titled *string* to the HelpBox, opening it if necessary. If the button is pressed, the procedure *id* is executed.

WRITE

statement ::= WRITE '(' expression (',' expression)* ')'

Writes a serie of expressions (of type CARD, INTEGER or STRING) in the HelpBox, opening it if necessary.

WHILE

statement ::= WHILE expression DO statement

The statement after DO is executed repeatedly as long as the Boolean expression is True.

The expression is evaluated before the statement is executed, so if the expression is False at the beginning, the statement is not executed at all.

PULL

statement ::= PULL expression [FROM stack_src] [TO stack_dst]
stack_src ::= *id*
stack_dst ::= *id*

Takes the last *n* (expression) cards of the source stack and add them at the end of the stack destination.

If stack_dst or stack_src is not specified then the context must specify them.

MOVE

statement ::= MOVE stack_interval TO stack_pos

Moves the contents of stack_interval to the stack_pos. The two stacks MUST be different.

Example :
A1=[13, 5, 8, 3, 2, 4]
A2=[1, 6, 9]

MOVE A1[2..4] TO A2[3];

A1=[13, 2, 4]
A2=[1, 6, 5, 8, 3, 9]

Interval

stack_interval ::= [*id*] range
stack_pos ::= [*id*] pos

Indicates an interval in a stack or a position. The allowed range is [0..!] for any stack.

If the stack (*id*) is not specified then the context must specify it.

range ::= '[' expression '..' expression ']' | '[' expression ']'
pos ::= '[' expression ']'

In range, if the second option is used, it describes a range of one card.

card_interval ::= expression .. expression | expression

Describes an interval of cards. For example, a royal flush in heart is : NINE+HEART .. KING+HEART.

TURN

statement ::= TURN stack_interval SIDE [UP | DOWN | SHADED]

Turn all cards in the interval on the specified side.

INVERSE

statement ::= INVERSE stack_interval

Inverse the order of the cards in the interval. Inversing an ascending sequence makes it a descending sequence.

ADD

statement ::= ADD card_interval [TO stack_dst]
stack_dst ::= *id*

Add specific cards to a stack.

REMOVE

statement ::= REMOVE stack_interval

Removes in interval of cards in a stack, destroying the cards.

SHUFFLE

statement ::= SHUFFLE [stack_dst]
stack_dst ::= *id*

Shuffles the cards in a stack.

BREAK

statement ::= BREAK [PROCEDURE]

BREAK is used to get out of a block (in a WITH or in a procedure). To get out of a procedure when in a WITH body, use BREAK PROCEDURE. A BREAK inside a WHILE body acts like a BREAK PROCEDURE.

RETURN

statement ::= RETURN expression

It is used to get out of a function or a predicate and to specify the return value.

:=

statement ::= left_value ':=' expression
left_value ::= *id* | stack_pos

Assign the value of the expression to the memory cell described by the left_value.

IF

statement ::= IF expression THEN statement [ELSE statement]

If the Boolean expression after IF is True, the statement after THEN is executed.

Otherwise, if the ELSE part is present, the statement after ELSE is executed.

FLASH

statement ::= FLASH stack_interval

Makes the cards in the interval flash three times.

BEGIN ... END

statement ::= BEGIN (statement)* END

When bracketed in this way, any number of consecutive statements can be treated as a single statement.

WITH

statement ::= WITH stack_dst DO statement FOR stack_src (',' stack_src)*
stack_src ::= *id*
stack_dst ::= *id*

WITH is an iterator construct. The statement is executed successively for each stack_src. stack_dst is used inside the statement to access the stack_src of the current iteration.

Files

***.cdl**
source files

***.cvc**
executable files

Include *.csg
saved player game files

References for CardsWorkShop

[1] Tarpel, C., Toutes les réussites et jeux de patiences, Guy Le Prat, 1975
[2] Brown, Douglas, 150 Solitaire Games, Harrow Books, 1972
[3] Les règlements officiels des jeux de cartes, International Playing Card Company Limited, 1977
[4] Morehead, Albert H., The pocket book of games, Pocket Books, 1944
[5] Berloquin, Pierre, Les réussites les plus passionnantes, Marabout, 1980
[6] Freha, Pierre, Jouer auz réussites, de Vecchi, 1991
[7] Bezanovska et Kitchevats, Le livre des patiences, Homme, 1987

[8] Wirth, Niklaus, Algorithm + Data Structures = Program
[9] Aho, Sethi, Ullman, Compilateurs, Principes, techniques et outils, InterEditions