TEMPO FOR WINDOWS VERSION 2

USER'S GUIDE

© 1996 Steve Gregory

September 6, 1996

Email: steve@cs.bris.ac.uk
WWW: http://www.cs.bris.ac.uk/~steve
Tel: +44 117 954 5142
Fax: +44 117 954 5208

Department of Computer Science
University of Bristol
Bristol BS8 1UB
England

Contents

1 INTRODUCING TEMPO FOR WINDOWS 1

1.1 THE TEMPO LANGUAGE 1
1.2 SCOPE OF THIS GUIDE 1
1.3 INSTALLING TEMPO FOR WINDOWS 2
1.4 RUNNING TEMPO FOR WINDOWS 2

2 ENTERING QUERIES 3

2.1 CORRECTING AND RECALLING QUERIES 3
2.2 A SIMPLE QUERY 3
2.3 MULTIPLE PROCESSES 4
2.4 MULTIPLE WINDOWS 5
2.5 CONFIGURING THE TRACE 5
2.6 ERRORS 6
2.7 DEADLOCK AND TERMINATION 6
2.8 COMPILING PROGRAMS 7
2.9 THE PAUSE DIALOG 7
2.10 MANUAL SCHEDULING 8
2.11 OTHER TRACE OPTIONS 10

3 TEMPO SYNTAX AND OPERATORS 11

3.1 SYNTAX 11
3.2 OPERATORS 13

4 TEMPO EXECUTION AND TRACE MESSAGES 15

5 FUNCTIONS 18

+(X,Y,Z) 18
-(X,Y,Z) 18
**(X,Y,Z) 18*
/(X,Y,Z) 18
//(X,Y,Z) 18
mod(X,Y,Z) 18
head(L,H) 18
tail(L,T) 18

6 TESTS 19

D1 == D2 19
D1 \== D2 19
N1 > N2 19
N1 < N2 19
N1 >= N2 19
N1 =< N2 19
atom(D) 19
integer(D) 19
float(D) 19
number(D) 19
atomic(D) 20
list(D) 20

7 SUMMARY OF COMMANDS 21

compile(Filename) 21

listing *21*
quit *21*
set *21*
set(compile,Onoff) *21*
set(schedule,Type) *22*
set(pause,Onoff) *22*
set(windows,N) *22*
set(error,Onoff) *22*
set(process,Onoff) *23*
set(execute,Onoff) *23*
set(enable,Onoff) *23*
set(last,Onoff) *24*
set(try_list,Onoff) *24*
set(constraint_set,Onoff) *24*
set(try,Onoff) *25*
set(constraint,Onoff) *25*
set(data,Onoff) *25*
set(communication,Onoff) *25*

8 RUNTIME ERROR MESSAGES 26

C, at top level failed *26*
all alternatives in disjunction failed *26*
event variable in bad context *26*
function, C, has undefined value *26*
test, C, has undefined value *26*

1 Introducing Tempo for Windows

1.1 The Tempo language

Tempo for Windows is an implementation of the Tempo programming language running on IBM PC compatible machines under Windows 3.1 (or later). Version 1 of Tempo for Windows supported the language described in the paper

> Tempo: a declarative concurrent programming language.
> Steve Gregory and Rafael Ramirez.
> In *Proc. of the International Conference on Logic Programming* (Tokyo, June),
> MIT Press, 1995, pp. 515–529.
> `http://star.cs.bris.ac.uk/papers/tempo.html`

Version 2 supports the new language described in the following paper:

> A declarative approach to concurrent programming.
> Steve Gregory.
> Technical Report.
> Department of Computer Science, University of Bristol, 1996.
> `http://star.cs.bris.ac.uk/papers/newtempo.html`

The new language is a *superset* of the original one: it has been extended to include data values and the ability to perform computation on them without sacrificing the benefits of the original language. The reader is therefore recommended to refer to *both* of the above papers to gain an understanding of the current language. Programs written for version 1 of Tempo for Windows should run unchanged under version 2 (the only differences being that the '<−', '+' and '<' operators have been replaced by '**:**−', '#' and '<<', respectively).

Tempo for Windows is implemented in Parlog for Windows, which may be obtained from

> Parallel Logic Programming Ltd.
> PO Box 49
> Twickenham TW2 5PH
> England
> Telephone: +44 1454 201652

1.2 Scope of this Guide

This Guide has four main sections. Section 2 describes the Tempo for Windows system, including how to get it running and how to run Tempo programs, with examples. Section 3 defines the syntax of the language and lists the predefined operators. Section 4 outlines how programs are executed, and what the trace messages mean. Sections 5 and 6 list the predefined function relations and test relations. Section 7 summarizes the available commands. Finally, Section 8 explains the most important error messages.

1.3 Installing Tempo for Windows

The Tempo for Windows package is distributed as a ZIP file containing the following files and directories:

- Five files which constitute the Tempo for Windows system: 'TEMPO.EXE', 'TEMPO.OVL', 'TEMPO.EXP', 'TEMPO.INI', 'WINMEM32.DLL'.

- Two directories, named 'EX1' and 'EX2', containing Tempo code which you can study. 'EX1' contains examples that were distributed with version 1 of Tempo for Windows, while those in 'EX2' are new. Some of these example programs will be used in what follows.

- The User's Guide 'MANUAL.DOC', which you are reading now.

1.4 Running Tempo for Windows

One way to run Tempo for Windows is to use the Windows File Manager to select the file 'TEMPO.EXE' and then choose its '`File/Run`' menu option, or just double-click on the file. Alternatively, you can create a program item for 'TEMPO.EXE', using the Program Manager's '`File/New`' option.

Tempo for Windows starts by displaying a graphical banner which remains on the screen while the system is loaded. When loading is complete, the banner is replaced by two text windows: the *main* window (named '`Tempo for Windows`'), which encloses the *console* window (named '`Console`'). The console window contains a textual banner, followed by a prompt '`> `'.

The console window acts rather like a standard DOS window: commands and Tempo queries can be typed in at the supervisor prompt. Unlike a DOS window, you can scroll back over text that has disappeared from view, cut and paste text, and reenter previous commands with a single keystroke. The main window provides two menus: '`Edit`' and '`Window`', which are quite standard.

2 Entering queries

The only way to perform any useful function is by typing something into the console window at the '> ' prompt. (Although Tempo for Windows also provides menus, these are only for the purposes of selecting windows and editing text.) Anything typed in the console window must be a *term* in the syntax used by logic programming languages such as Prolog or Parlog; i.e., it must take the form

name(*arg*$_1$,*arg*$_2$,...,*arg*$_n$).

Note that the terminating period character is essential. A few *infix* operators are also allowed in terms, e.g., '#', '<<', '**;**'; see Section 3 for a full description of Tempo syntax.

Each term typed into the console window may be either a command or a query:

- Any term of one of the following forms is treated as a *command*:

```
quit.
listing.
set.
set(arg1,arg2).
compile(arg1).
```

- Any other term is treated as a *query*.

Probably the most important command is `quit`, which provides the only way to exit from Tempo for Windows.

2.1 Correcting and recalling queries

If you make a mistake while entering a command or query, it can easily be corrected at any time before you type `Return`: just use the mouse or cursor keys to position the cursor, and the `Backspace` or `Delete` keys to delete characters. Arbitrary text, from the console window or elsewhere, can be pasted in and edited while forming a query: the '`Edit`' menu and '`Window`' menu are useful for this purpose.

It is even possible to recall the last (or any previous) query, avoiding the need to type it in again. Simply move the cursor to the appropriate line in the console window and type `Return`: that line will then be copied to the bottom of the console window and executed automatically. If desired, the line can be edited before typing `Return`.

2.2 A simple query

As a simple first problem for Tempo for Windows to solve, try entering a query containing a few primitive constraints. Type the following query into the console window (remembering to type the period, and then `Return`, at the end of the query):

```
H << E, E << T.
```

The query is solved by executing three events `H`, `E`, and `T`, in that order.

A trace of the execution is displayed in the console window, as follows:

```
root: Starts with events []
root: Enable H
root: Execute H
root: Enable E
root: Execute E
root: Enable T
root: Execute T
root: Terminates
```

The label '`root:`' on each line identifies the process from which the trace message comes. In this example the only process is the initial one, which is named `root`. In general, processes may be spawned dynamically: these will have different names, as explained in Section 2.3. The first and last lines of the trace simply report the invocation and termination of this solitary process. The remaining lines show when each of the three events is *enabled* and *executed*: each of `H`, `E`, and `T` is enabled and then immediately executed.

2.3 Multiple processes

In the above example, each event can be executed as soon as it is enabled, so why does the trace show the enabling and execution of an event as separate messages? The reason is that, whenever an event is shared by several processes, it has to be enabled by *all* of them before it can be executed. The trace therefore shows when a process has *enabled* the event, and when a process detects that the event has been *executed*, which may be much later or not at all.

To illustrate this situation, type the query

```
H << E, {E << T}.
```

which is the same as the previous one except that the two constraints are to be solved in separate processes: `H<<E` will be solved in the `root` process, while `E<<T` is solved in a process named `root.1`. (In general, the *N*th process spawned by process *P* is named *P*`.`*N*.) The observable behaviour of a query (i.e., the order in which events are executed) is determined *only* by the constraints it contains, and *not* by the way in which they are allocated to processes. In this example, `H`, `E`, and `T` must still be executed in that order.

When you run the above query the following trace is displayed in the console window:

```
root: Starts with events []
root.1: Starts with events [E]
root: Enable H
root.1: Enable E
root: Execute H
root: Enable E
root.1: Execute E
root.1: Enable T
root.1: Execute T
root.1: Terminates
root: Execute E
root: Terminates
```

The second line of the trace reports the spawning of the process named `root.1`: the message "`Starts with events [E]`" means that event `E` is shared between the new process and the process (`root`) that spawned it. Every shared event must be enabled by all

processes that share it before it can be executed: this is why each of our two processes reports "`Enable E`" when it enables `E`. Similarly, both processes report "`Execute E`" after `E` has actually been executed. This should not be taken to mean that `E` is executed twice! An event can only be executed once, but the exact time of execution cannot be known by any one process. The message "`Execute E`" displayed by a process therefore only means that that process has *observed* that `E` has been executed.

2.4 Multiple windows

Whenever there is more than one process, their trace messages are interleaved in the console window. This can easily become hopelessly confusing, but there is a simple solution. Type the command

```
set(windows,2).
```

This asks the system to trace each process in a separate window; these are laid out in tiled fashion on the screen, each window occupying the full width of the screen. Then repeat the previous query: each process will now be traced in a separate window. Just before a process terminates, a dialog will appear with two buttons, labelled '`OK`' and '`Quit`'. This is to give you a chance to examine that process's trace before its window is deleted. Click the '`OK`' button to proceed. The dialog is the same as the pause dialog, described in Section 2.9.

The second argument of `set` specifies the number of windows (>= 1) per screen: this does *not* limit the number of windows that can be created, only the number that are visible at any time. In the present example, two windows per screen is sufficient because there are only two processes. If there will be a large number of processes, you will need to (a) increase the number of windows per screen or (b) expect windows to overlap other windows. This is not necessarily a problem, as it is possible to switch to any trace window at any time in order to inspect its contents; this is especially useful in conjunction with the *trace pause* and *manual scheduling* facilities, described in Sections 2.9 and 2.10, respectively.

To resume tracing in the console window, type

```
set(windows,0).
```

The `windows` option is one of several that can be changed by the `set(_,_)` command. To check the current settings of *all* of the options, type the command

```
set.
```

Some of the other options available will be illustrated in what follows; all of them are described in Section 5.

2.5 Configuring the trace

If you find that the trace presents too much detail, you can turn off the display of specific trace messages by the `set(_,_)` command. For example, to see only when events are executed, you can turn off reporting of all other actions by typing the commands

```
set(enable,off).
set(process,off).
```

Then the query

```
H << E, E << T.
```

gives the following trace:

```
root: Execute H
root: Execute E
root: Execute T
```

Tracing may be turned on again by a command such as

```
set(enable,on).
```

2.6 Errors

Instead of *terminating* successfully, a Tempo query may result in *non-termination*, *deadlock*, or *error*.

A common error is an attempt to solve an undefined constraint, as in the query

```
fred(X).
```

which might cause the enigmatic message 'ERROR' to be displayed. This usually means that you have forgotten to compile the Tempo program in which the constraint (`fred`) is defined. Compiling programs is discussed in Section 2.8. Curiously, no error results from attempting to solve a constraint that has no arguments (e.g., `fred`); this is because such a constraint is trivially solved regardless of the definition of `fred`.

There are some other errors that produce the same error message, but most of them (for example, syntax errors) cause a self-explanatory message to be displayed. The most important error messages are listed in Section 8.

2.7 Deadlock and termination

A key feature of Tempo is that deadlock and termination are conceptually identical. This will occur when all query events, *except* any events `E` such that `E<<E`, have been executed: an event that precedes itself can clearly never be executed. For example, the query

```
A << B, B << A.
```

will immediately terminate without executing any events. Now try the query

```
A << B, {B << A}.
```

which is logically identical, but has the constraints solved in separate processes. Again, no events will be executed, but this time the system will *deadlock*. The trace shows that the `root` process enables `A` while the `root.1` process enables `B`, but then each process waits in vain for the other to enable the same event. The difference between termination and deadlock is only one of implementation: given a deadlock detection algorithm, a deadlocked program could be made to terminate.

You can break the deadlock by hitting the `Ctrl-break` key. This will cause the query

to be aborted immediately and a `stopped` message displayed.

Deadlock is just one reason why a query may fail to terminate. It is possible for a query to run indefinitely, but still produce useful results. In such cases, you will also need to use the `Ctrl-break` key to regain control of your machine.

2.8 Compiling programs

So far we have considered queries that contain only primitive constraints. To make use of user-defined constraints it is first necessary to load the Tempo program containing the definitions of the constraints used. This is done by the `compile(_)` command.

As an example we shall compile the definition of the `phil` constraint which is described in *Tempo: a declarative concurrent programming language*, and is provided in the distribution as file 'PHIL.TEM'. The constraint `phil(H,E,T)` represents a "philosopher" that repeatedly gets hungry (`H`), eats (`E`), and thinks (`T`),. Type the command

```
compile('ex1\phil.tem').
```

(the exact filename will depend on your current directory).

Having done this, you can check that the code is loaded correctly, by the command

```
listing.
```

which shows the source code of the loaded program.

The `compile(_)` command generates "object code", which is an internal representation of the program, slightly different from the source code. The differences are that many constraints are labelled with a list of events, and some extra constraints (implicit in the source program) are added to the code. For example, an explicit `E<<E#1` constraint is added for every event `E#1` that appears in the body of a constraint definition. Also, a `0<<E` constraint is added for every event `E` in the body that appears neither in the head nor on the right of a '<<' constraint, and for every event in a process that does not appear on the right of a '<<' constraint. To view this internal form, type the command

```
set(compile,on).
```

before compiling the program.

2.9 The pause dialog

Having compiled the philosopher example, try running the query

```
phil(H,E,T).
```

The trace will show a sequence of events `H`, `E`, `T`, `H#1`, `E#1`, `T#1`, etc., which never ends:

```
root: Starts with events []
root: Enable H
root: Execute H
root: Enable E
root: Execute E
root: Enable T
root: Execute T
root: Enable H # 1
root: Execute H # 1
root: Enable E # 1
root: Execute E # 1
root: Enable T # 1
root: Execute T # 1
root: Enable H # 1 # 1
...
```

In this kind of situation, it may be desirable to slow down the execution so that the trace messages can be viewed before they disappear from the screen. This can be done using a *pause dialog*. Type the command

```
set(pause,on).
```

Now, each time a trace message is displayed, a dialog will appear with two buttons, labelled 'OK' and 'Quit'. Click the 'OK' button to continue with the execution, or the 'Quit' button to abort the execution of the query. Before clicking either button, you can select other windows (e.g., trace windows or the console) to examine their contents.

(Unfortunately, the pause dialog will disappear behind the main Tempo window if you click on any other window while the dialog is on display. To retrieve the dialog: minimize the main Tempo window, drag the dialog to a clear part of the screen, e.g., the bottom, and then restore the Tempo window.)

2.10 Manual scheduling

In general, a Tempo query specifies a *partial* order of events: whenever the query is run, the Tempo interpreter can generate *any* execution sequence that satisfies the constraints in the query. For programs that use disjunction, a given query does not even uniquely determine the *constraints* that need to be satisfied. In short, Tempo execution is *non-deterministic*.

In a real program, we don't care which execution sequence is generated by a particular run, provided that it is correct. In any case, the non-determinism of a constraint is often reduced by conjoining it with other constraints, possibly in other processes. However, during program development, it is useful to be able to test individual constraints in isolation.

Normally, if the execution is non-deterministic, Tempo for Windows makes an arbitrary choice of which execution sequence to follow. The *manual scheduling* option allows the user to choose this interactively; in effect, the user can act like an extra constraint conjoined to the query.

To see how manual scheduling works, compile the file 'MORTAL.TEM' by typing the command

```
compile('ex1\mortal.tem').
```

The constraint `mortal_phil(H,E,T,D)` represents a philosopher that repeatedly thinks,

gets hungry, and eats, but may die (event D) at any time while thinking:

```
mortal_phil(H,E,T,D) :-
   phil(H,E,T),
   mortal(D,H,T).

mortal(D,H,T) :-
   H << H;
   T << D, mortal(D,H#1,T#1).
```

If you type the query

```
mortal_phil(H,E,T,D).
```

the trace will show that the philosopher eats once and then dies (event D):

```
root: Starts with events []
root: Enable H
root: Execute H
root: Enable E
root: Execute E
root: Enable T
root: Execute T
root: Enable D
root: Execute D
root: Terminates
```

Now turn on manual scheduling by typing

```
set(schedule,manual).
```

and run the query again. Now, before any event E is enabled, the system will show an *enable dialog* asking "Enable E?"; this has three buttons, labelled 'Yes', 'No', and 'Quit'. Clicking the 'Yes' button will tell this process to enable E. (If you always click 'Yes', the result will be the same as that of automatic scheduling.) If you click 'No', the system will present a dialog asking whether to enable another event that could be enabled, if any exist; if there are no other such events, the system will immediately repeat the question "Enable E?".

For the mortal philosopher example, the first dialog will ask "Enable H?". If you click 'No', the system will ask "Enable D?". If you again click 'No', the "Enable H?" dialog will reappear. This will go on indefinitely, showing you that only H or D can be enabled. Clicking 'Yes' to the question "Enable D?" will cause D to be enabled and executed and the process will terminate. Otherwise, if you answer 'Yes' to the question "Enable H?", the sequence of events H, E, T will be executed, and then there will be another choice, this time between H#1 and D.

(Like the pause dialog, the enable dialog will disappear behind the main Tempo window if you click on any other window while the dialog is on display. To retrieve the dialog: minimize the main Tempo window, drag the dialog to a clear part of the screen, e.g., the bottom, and then restore the Tempo window.)

2.11 Other trace options

We have already mentioned some of the messages that can appear in the trace output, and the use of the `set(_,_)` command to turn them on and off. There are actually 10 different types of options controlling trace messages. These options, and their default settings, are listed below and fully described in Sections 4 and 5:

```
process              on
enable               on
execute              on
last                 off
try_list             off
constraint_set       off
try                  off
constraint           off
data                 off
communication        off
```

3 Tempo syntax and operators

3.1 Syntax

Tempo syntax is defined here in a variant of BNF. Terminal symbols are enclosed in single quotes; non-terminal symbols are alphanumeric strings with upper case initials. The metasyntactic constructs used are:

|	disjunction
[*item*]	0 or 1 occurrences of *item*
{ *item* }	0 or more occurrences of *item*

A Tempo constraint definition is defined as `Definition`, while a Tempo query is a `Body`:

```
Definition          = Head ':-' Body

Body                = Body_constraint
                      { ',' Body_constraint }

Body_constraint     = '{' Process '}'
                    | Local_constraint

Process             = Local_constraint
                      { ',' Local_constraint }

Local_constraint    = '(' Disjunction ')'
                    | Top_constraint

Disjunction         = Alternative ';' Alternative
                      { ';' Alternative }

Alternative         = Disj_constraint
                      { ',' Disj_constraint }

Top_constraint      = Temporal_constraint
                    | Function_constraint

Disj_constraint     = Temporal_constraint
                    | Test_constraint

Temporal_constraint = User_constraint
                    | Prim_constraint

Head                = Temporal_name
                      '(' Var { ',' Var } ')'

User_constraint     = Temporal_name
                      '(' Arg { ',' Arg } ')'

Prim_constraint     = '0' '<<' Event
                    | Event '<<' Event
                    | Event '^:=' Data
```

```
Function_constraint = Eval_function_name
                      '(' Data { ',' Data } ')'

Test_constraint     = Test_name
                      '(' Data { ',' Data } ')'

Arg                 = Event
                    | Data

Var                 = Event_var
                    | Data_var

Event               = Event_var
                    | Event '#' Integer

Data                = Data_var
                    | Event '^'
                    | Number
                    | Constant
                    | Function_name
                      '(' Data { ',' Data } ')'

Function_name       = Eval_function_name
                    | Structure_name
```

The remaining symbols used in the above BNF are:

`Temporal_name`
`Test_name`
`Eval_function_name`
`Structure_name`
	Alphanumeric identifier beginning with a lower case letter.

`Event_var` `Data_var`	Alphanumeric identifier beginning with an upper case letter.

`Number`
`Integer`
`Constant`	

The three different types of constraint name (`Temporal_name`, `Test_name`, and `Eval_function_name`) form disjoint sets. There is a predefined set of test relations (`Test_name`) and a disjoint predefined set of function relations (`Eval_function_name`); these are listed in Section 6. Any other name that appears as a constraint name is taken to be a `Temporal_name`.

There are two types of function name that can appear in data values: `Eval_function_name` and `Structure_name`. These are distinguished in the same way: any function name that is not one of the predefined set of function relations is taken to be a `Structure_name`.

The distinction between the two types of variable (`Event_var` and `Data_var`) is made by a type analysis that takes into account the declared types of temporal relations. Type declarations in a Tempo program take the form of a `Type_dec`, that indicates whether

each argument is of type `Event` ('e') or `Data` ('d'):

```
Type_dec          = 'type' Temporal_name
                    '(' Type { ',' Type } ')'

Type              = 'e' | 'd'
```

As an alternative to the standard notation $f(t_1, ..., t_n)$, constraints and functions with one or two arguments can be written using prefix, infix, or postfix operators. The set of available operators is predefined and is listed in Section 3.2 below.

3.2 Operators

The following table lists the predefined operators that are used in Tempo for Windows. These include operators used to construct constraint definitions and type declarations, and for primitive temporal relations, functions (e.g., '+', '*', etc.), and tests (e.g., '<', '==', etc.). The operators '`where`' and '\' are used only for output.

Precedence	Type	Operator
1200	xfx	`:-`
1100	xfy	`;`
1050	fx	`type`
1050	xfy	`where`
1000	xfy	`','`
700	xfx	`<<`
700	xfx	`^:=`
700	xfx	`==`
700	xfx	`\==`
700	xfx	`>`
700	xfx	`<`
700	xfx	`>=`

Precedence	Type	Operator
700	xfx	`=<`
700	xfx	`<<*`
500	yfx	`+`
500	yfx	`-`
400	yfx	`*`
400	yfx	`/`
400	yfx	`//`
300	xfx	`mod`
200	xf	`^`
100	yfx	`#`
50	yfx	`\`

There are many other predefined operators which are inherited from the underlying system. These are not used by Tempo for Windows but are available for use in Tempo programs if required:

Precedence	Type	Operator
1200	fx	`:-`
1200	fx	`?-`
1200	xfx	`-->`
1150	fx	`meta_predicate`
1150	fx	`volatile`
1150	fx	`dynamic`
1150	fx	`multifile`
1150	fx	`initialization`
1150	fx	`public`
1100	xfy	`'|'`
1075	fx	`<-`
1075	xfx	`<-`
1050	xfy	`->`
1050	fx	`depth`
1050	fx	`mode`
1050	fx	`prolog`
1050	fx	`database`
1050	fx	`optimize`
1050	fx	`debuggable`
1050	fx	`version`
1050	xfy	`::::`
1050	xfy	`:`
1050	xfy	`::`
1050	xfy	`:::`
1025	xfx	`when`
1025	xfy	`&`

Precedence	Type	Operator
1010	xfx	`@=`
900	fx	`one`
900	fy	`nospy`
900	fy	`not`
900	fy	`spy`
900	fy	`\+`
850	yfx	`~>`
850	yfx	`<~`
700	xfx	`=\=`
700	xfx	`=:=`
700	xfx	`@>`
700	xfx	`@<`
700	xfx	`@>=`
700	xfx	`@=<`
700	xfx	`=..`
700	xfx	`is`
700	xfx	`=`
700	xfx	`\=`
700	xfx	`<=`
500	fx	`+`
500	fx	`-`
500	yfx	`\/`
500	yfx	`/\`
400	yfx	`>>`
10	xf	`?`

4 Tempo execution and trace messages

To understand the meaning of Tempo's trace messages, it is necessary to know something about the way in which Tempo programs are executed. This is the subject of this section.

Each Tempo process executes an interpreter, which maintains two data structures: a *constraint set*, containing the constraints still to be satisfied, and a *try list*, a queue of events (i.e., variables) that have not yet been executed but need to be tried. The interpreter repeatedly executes a loop which is driven by receiving an event that has been executed (as a result of having been enabled by all processes); the process terminates when all events that it has enabled have been executed. The interpreter works as follows:

1. If the `process` trace option is on, the tracer displays the message

   `Starts with events [`S_1`,...,`S_n`]`

   where $S_1, \ldots, S_n$ are the events shared with the process that spawned this process ($n = 0$ for the initial, `root`, process).

2. All events appearing in the process's constraints are placed in the try list.

3. If any *process constraints* (constraints of the form `{_}`) appear in the constraint set, new processes are spawned to solve them. Any events that are shared with such spawned processes are also added to the try list.

4. If the `try_list` trace option is on, the tracer displays the try list `[`E_1`,...,`E_n`]` in a message

   `Try List: [`E_1`,...,`E_n`]`

5. The first event E_1 is removed from the try list and tested against each of the constraints in the constraint set. If E_1 appears in the arguments of a user-defined constraint, that constraint is expanded, and events that appear only in its body are added to the rear of the try list.

6. If the `constraint` trace option is on, for each constraint C, the tracer displays the message

   `Try constraint `E_1`, `C`, `R

   where R is the result of trying E_1 against constraint C: `enabled`, `disabled`, `cond_enabled` (conditionally enabled), `wait_others`, `need_value`, `constrained`, or `assign_enabled`.

   While trying E_1 against a disjunction that includes a test C (provided that C does not include the value of any events other than C), C is evaluated. If the `data` trace option is on, the tracer displays the message

   `Test `C

   At various times, function constraints whose outputs appear in data arguments of other constraints are evaluated. If the `data` trace option is on, the tracer displays the message

```
Function C
```

7. If the data value of E_1 was needed while trying to enable E_1 (because E_1 appears in tests in a disjunction), any assignments to E_1 are done. If the `data` trace option is on, the tracer displays a message such as

   ```
   Assign E1 ^:= V
   ```

   for each assignment.

8. If the `try` trace option is on, the tracer displays the message

   ```
   Try E1, R
   ```

   where R is the result of trying to enable E_1: `enabled`, `disabled`, `cond_enabled` (conditionally enabled), or `wait_others`.

9. If the `constraint_set` trace option is on, the tracer displays the message

   ```
   Constraint Set: [C1,...,Cn]
   ```

   where `[`C_1`,...,`C_n`]` is the current constraint set.

10. If the result of trying to enable E_1 was `wait_others`, i.e., the event was used in a disjunction *and* the other processes that share it have not enabled it:

    10a. A new local event (named `other(`E_1`)`) is created — this will be executed when all other processes have enabled E_1 — and the constraint `other(`E_1`)<<`E_1 is added to the constraint set.

    10b. If the `last` trace option is on, the tracer displays the message

    ```
    Creating constraint other(E1) << E1
    ```

    In this case, the tracer will also show when the local event `other(`E_1`)` is executed: some time after all other processes have enabled E_1.

    Otherwise

    If the scheduling option is set to manual and the user replies 'No' to the enable dialog that is displayed, asking "Enable E_1?":

    10c. The event E_1 is deferred.

    Otherwise

    10d. Any assignments to E_1 are done. If the `data` trace option is on, the tracer will display the trace message

    ```
    Assign E1 ^:= V
    ```

    for each such assignment.

10e. E_1 is marked as enabled. It will then be executed after it is enabled by any other processes that share it.

10f. If the `enable` trace option is on, the tracer displays the message

```
Enable E1
```

11. If the try list is not empty, the loop is repeated from step 4.

12. If the process has enabled some events that have not yet been executed:

13. The process receives an event *E* that has just been executed.

14. If the `execute` trace option is on, the tracer displays the message

```
Execute E
```

15. All constraints of the form *E*<<*R* are removed from the constraint set, and *R* is added to the rear of the try list, as well as any deferred events (see step 10c).

16. Any tests *C* on *E* that exist in the constraint set (having been promoted from disjunctions) are evaluated. If the `data` trace option is on, the message

```
Test C
```

is displayed. The test should succeed; otherwise a fatal error occurs.

17. The loop is repeated from step 3.

Otherwise

98. If the `process` trace option is on, the tracer displays the message

```
Terminates
```

99. The process terminates.

5 Functions

This section lists all of the function relations that are predefined in Tempo for Windows. The last argument of each of them is the output argument, so they can all be written using functional notation in data value positions in programs. E.g., the terms

```
+(X,Y)   -(X,Y)   *(X,Y)   /(X,Y)   //(X,Y)  mod(X,Y)
head(L)  tail(L)
```

are interpreted as evaluable functions. Since '+', '−', '*', '/', and '//' are all infix operators, those terms may also be written as

```
X+Y      X-Y      X*Y      X/Y      X//Y     X mod Y.
```

+(X,Y,Z)

Produces a value for `Z`: `X + Y`.

-(X,Y,Z)

Produces a value for `Z`: `X − Y`.

***(X,Y,Z)**

Produces a value for `Z`: `X * Y`.

/(X,Y,Z)

Produces a value for `Z`: `X / Y`.

//(X,Y,Z)

Produces a value for `Z`: `X // Y`, i.e., the greatest integer that is less than or equal to the quotient `X / Y`.

mod(X,Y,Z)

Produces a value for `Z`: `X mod Y`, i.e., the remainder of dividing `X` by `Y`.

head(L,H)

Produces a value for `H`: the head of list `L`.

tail(L,T)

Produces a value for `T`: the tail of list `L`.

6 Tests

This section lists all of the test relations that are predefined in Tempo for Windows.

D1 == D2

Succeeds if data values D1 and D2 are identical.

D1 \== D2

Succeeds if data values D1 and D2 are not identical.

N1 > N2

Succeeds if data values N1 and N2 are numbers and N1 is greater than N2. A runtime error will result if N1 and N2 are not numbers.

N1 < N2

Succeeds if data values N1 and N2 are numbers and N1 is less than N2. A runtime error will result if N1 and N2 are not numbers.

N1 >= N2

Succeeds if data values N1 and N2 are numbers and N1 is greater than or equal to N2. A runtime error will result if N1 and N2 are not numbers.

N1 =< N2

Succeeds if data values N1 and N2 are numbers and N1 is less than or equal to N2. A runtime error will result if N1 and N2 are not numbers.

atom(D)

Succeeds if data value D is an atom (constant).

integer(D)

Succeeds if data value D is an integer.

float(D)

Succeeds if data value D is a floating point number.

number(D)

Succeeds if data value D is a number, either integer or floating point.

`atomic(D)`

Succeeds if data value `D` is either an atom or a number.

`list(D)`

Succeeds if data value `D` is a non-empty list. (An empty list is the *atom* '`[]`'.)

7 Summary of commands

`compile(Filename)`

`Filename` must be the name of a file containing a Tempo program. The filename is interpreted relative to the current directory and must include the file extension, if any (the extension '.TEM' is recommended).
This command loads the Tempo program contained in the file into memory, checks it for errors, and converts it into an internal form. Any obvious errors in the program will be reported.
Note that the loaded program *replaces* any program currently in memory, so the file must be self-contained, containing the definitions of all constraints that it uses.
If the `compile` option is on, the internal form of the program is displayed during compilation.

`listing`

Displays the source form of the entire program currently in memory.

`quit`

Exits from Tempo for Windows.

`set`

Displays the current settings for each of the options. These settings can be changed by the `set(_,_)` command, described below. The default settings are:

```
compile = off
schedule = auto
pause = off
windows = 0
error = on
process = on
execute = on
enable = on
last = off
try_list = off
constraint_set = off
try = off
constraint = off
data = off
communication = off
```

`set(compile,`*`Onoff`*`)`

Tells the compiler to display the internal form of the program during compilation (if *`Onoff`* is `on`) or not (if *`Onoff`* is `off`).

set(schedule,*Type*)

Sets the scheduling strategy to automatic (if *Type* is auto) or manual (if *Type* is manual).
With automatic scheduling, whenever a process can enable an event it will do so, regardless of whether any other events can be enabled.
With manual scheduling, the system will always ask the user before enabling an event. Before any event E is enabled, an *enable dialog* (with three buttons, labelled 'Yes', 'No', and 'Quit') will ask "Enable E?";. Clicking the 'Yes' button will tell this process to enable E, as usual. Clicking 'No' will cause the event to be deferred, and the system will then ask the user to enable each of the other events, if any, that could be enabled. Clicking 'Quit' will abort the execution.
(The enable dialog will disappear behind the main Tempo window if any other window is clicked while the dialog is on display. To retrieve the dialog: minimize the main Tempo window, drag the dialog to a clear part of the screen, e.g., the bottom, and then restore the Tempo window.)

set(pause,*Onoff*)

Turns the trace pause option on (if *Onoff* is on) or off (if *Onoff* is off).
If this option is on, a *pause dialog* will appear each time a trace message is displayed. The dialog has two buttons, labelled 'OK' and 'Quit'. If the 'OK' button is clicked, the execution continues; if 'Quit' is clicked, the execution will be aborted. Before clicking any button, other windows (e.g., trace windows or the console) can be selected and viewed at leisure.
(The pause dialog will disappear behind the main Tempo window if any other window is clicked while the dialog is on display. To retrieve the dialog: minimize the main Tempo window, drag the dialog to a clear part of the screen, e.g., the bottom, and then restore the Tempo window.)

set(windows,*N*)

Selects whether trace output is to be displayed in multiple windows, and sets the size of those windows.
If *N* is 0, trace output from all processes will appear in the console window.
If *N* is a positive integer, trace output from each process will appear in a separate, dynamically created, window. The height of the trace windows, laid out in tiled fashion, is such that *N* windows can be viewed at the same time. Their width is approximately the width of the screen.

set(error,*Onoff*)

Selects the display of runtime error messages (if *Onoff* is on) or suppresses them (if *Onoff* is off).
A full list of these runtime error messages appears in Section 8.

set(process,*Onoff*)

Turns the `process` trace option on (if *`Onoff`* is `on`) or off (if *`Onoff`* is `off`). This controls the display of both `start` and `terminate` trace messages.
If this option is on, a trace message will be displayed each time a process is spawned:

`Starts with events [`E_1`,...,`E_n`]`

where $E_1, \ldots, E_n$ are the events that are shared between the new process and the process that spawned it, and a further message will be displayed when the process terminates:

`Terminates`

set(execute,*Onoff*)

Turns the `execute` trace option on (if *`Onoff`* is `on`) or off (if *`Onoff`* is `off`).
If this option is on, a trace message will be displayed each time a process executes an event. The message takes the form

`Execute` *`E`*

where *`E`* is the event being executed.
Note that, if `E` is a shared event, this trace message will be displayed by all of the processes that share it. This does *not* mean that the event is executed more than once: rather, the message should be read as "this process has observed that `E` has been executed".

set(enable,*Onoff*)

Turns the `enable` trace option on (if *`Onoff`* is `on`) or off (if *`Onoff`* is `off`).
If this option is on, a trace message will be displayed each time a process enables an event. The message takes the form

`Enable` *`E`*

where *`E`* is the event being enabled.

set(last,*Onoff*)

Turns the `last` trace option on (if *Onoff* is `on`) or off (if *Onoff* is `off`).
If this option is on, a trace message will be displayed if an event *E* is used in a disjunction in this process but is not known to have been enabled by the other processes that share it. In this case, the process must wait for the other processes to enable *E* before it can do so itself.
The message takes the form

```
Creating constraint other(E) << E
```

This means that a local event named `other(`*E*`)` is dynamically created — this will be executed when all other processes enable *E* — and a constraint `other(`*E*`)` <<*E* is added to the process's constraint set. These events and constraints will appear in the other trace messages.
If the `last` trace option is on, the execution of the local event will also be traced, by a message

```
Execute other(E)
```

set(try_list,*Onoff*)

Turns the `try_list` trace option on (if *Onoff* is `on`) or off (if *Onoff* is `off`).
If this option is on, a trace message will be displayed showing the contents of the try list each time that a process tries to enable an event.
The message takes the form

```
Try List: [E1,...,En]
```

where $E_1, \ldots, E_n$ are the events in the try list.

set(constraint_set,*Onoff*)

Turns the `constraint_set` trace option on (if *Onoff* is `on`) or off (if *Onoff* is `off`).
If this option is on, a trace message will be displayed showing the contents of the constraint set each time that a process tries to enable an event.
The message takes the form

```
Constraint Set: [C1,...,Cn]
```

where $C_1, \ldots, C_n$ are the constraints in the constraint set.

set(try,*Onoff*)

Turns the `try` trace option on (if *`Onoff`* is `on`) or off (if *`Onoff`* is `off`).
If this option is on, a trace message will be displayed showing the result of each attempt by a process to enable an event.
The message takes the form

```
Try E, R
```

where *`E`* is the event being tried and *`R`* is the result of trying to enable *`E`*: `enabled`, `disabled`, `cond_enabled` (conditionally enabled), or `wait_others`.

set(constraint,*Onoff*)

Turns the `constraint` trace option on (if *`Onoff`* is `on`) or off (if *`Onoff`* is `off`).
If this option is on, a trace message will be displayed showing the result of testing an event against each constraint in the constraint set, while a process is trying to enable an event.
The message takes the form

```
Try constraint E, C, R
```

where *`E`* is the event being tried, *`C`* is the constraint, and *`R`* is the result of testing *`E`* against constraint *`C`*: `enabled`, `disabled`, `cond_enabled` (conditionally enabled), `wait_others`, `need_value`, `constrained`, or `assign_enabled`.

set(data,*Onoff*)

Turns the `data` trace option on (if *`Onoff`* is `on`) or off (if *`Onoff`* is `off`).
If this option is on, trace message will be displayed to show operations involving data values, i.e., assignments, functions, and tests.
These messages take the form

```
Assign E ^:= V

Test C

Function C
```

set(communication,*Onoff*)

Turns the `communication` trace option on (if *`Onoff`* is `on`) or off (if *`Onoff`* is `off`).
If this option is on, trace messages will be displayed to show the low-level communication between processes for transferring data values of events.

8 Runtime error messages

Most of Tempo for Window's error messages are self-explanatory, but there are a few that are specific to Tempo; these are explained below.

C, at top level failed

This indicates that a test constraint, *C*, in the constraint set has failed.
Test constraints are allowed only inside disjunctions and are usually evaluated before the disjunction is used. However, if a test constraint includes the value of an event *E*, it may remain in the disjunction and subsequently be promoted to the top-level constraint set, where it will remain until *E* is executed. If the constraint then fails, this fatal error is the result.

all alternatives in disjunction failed

This indicates that test constraints have failed in *all* alternatives of a disjunction.

event variable in bad context

This fatal error may occur while trying to enable an event variable *E*. It indicates that the interpreter cannot determine whether or not *E* is enabled, because it would need to either expand a user-defined constraint inside a disjunction or assign a data value to *E* inside a disjunction, neither of which is possible.

function, *C*, has undefined value

This fatal error indicates the evaluation of a function constraint *C* that has one or more undefined input arguments. This error can only occur if *C*'s input arguments include the data value of an event (*E*^) such that *E* has been executed without being given a value.

test, *C*, has undefined value

This fatal error indicates the evaluation of a test constraint *C* with one or more undefined arguments. This error can only occur if *C*'s input arguments include the data value of an event (*E*^) such that *E* has been executed without being given a value.