WSL

Programmer's Reference Manual

Dr Martin Ward Feb. 2002;
revised T. Hardcastle Nov. 2003 and Stefan Natelberg Jul. 2008

Table of Contents

Section 1 : Introduction to FermaT/WSL

1.0: Preface

It is assumed in this document that the reader is already familiar with programming in imperative languages. Anyone who is not so familiar has probably downloaded this file by accident.

1.1: About WSL

WSL is a Wide Spectrum Language. That is to say that the language contains a wide selection of possible structures, so as to make it as easy as possible to translate a program or algorithm into WSL.

The "Spectrum" in "Wide Spectrum Language" refers to the range of operations, from "low level" things such as assignments, IF statements and gotos to high level operations such as specification statements. The highest level WSL operations are not implemented in FermaT (because they cannot be implemented on *any* machine!)

Besides having all the usual programming structures and commands, WSL also contains commands, functions and routines for operating on programs written in WSL. This gives the user a tool for analyzing, rewriting, and simplifying programs, and indeed for writing programs which analyze and rewrite programs. This is the main purpose of WSL.

Further resources and information on FeramT/WSL can be found at

http://www.cse.dmu.ac.uk/~mward/fermat.html

or

http://www.dur.ac.uk/martin.ward/fermat.html

When FermaT is revised, the new release is made available from these sites.

1.2: Getting started

To unpack the tar file and build FermaT:

```
gunzip < fermat3.tar.gz | tar xf -
cd fermat3
source DOIT.csh    (OR "source DOIT.sh" depending on your shell)
make test
```

If for any reason the tests should fail, please report this to Martin.Ward@durham.ac.uk

Once the program has installed, various commands should be available from the command line interface of your OS, which will compile, run, or rewrite programs.

Each time you log in you will need to source the DOIT.csh or DOIT.sh script unless you include these, or similar commands in your .profile or .cshrc file.

The program files on which the program operates are simply plain text files containg program listings, with the suffix .wsl

```
wsl FILENAME.wsl
```

will compile and run the file FILENAME.

```
wsl FILENAME
```

will have the same effect --- by default, the compiler expects that your file will have a .wsl suffix.

```
dotrans FILENAME1.wsl FILENAME2.wsl Transformation
```

will apply a transformation called Transformation to the file FILENAME1.wsl and will put the result in the file FILENAME2.wsl. A list of the available transformations and their uses will appear later in this manual.

```
wsl2scm FILENAME1.wsl FILENAME2.scm
```

will translate a WSL file called FILENAME1.wsl into a Scheme program called FILENAME2.scm

1.3: Contents of the distribution

Here is a brief description of the files in this distribution:

File(s)	Description
DOIT.csh DOIT.sh	Source one of these files into your shell to set up your environment variables and path to use FermaT.
LICENSE	A copy of the GNU General Public License
Makefile	Type "make test" to build and test FermaT
bin/*	Perl scripts to build and run FermaT
config/*	Perl modules and other support files
example/*	Example files: how to write a transformation
doc/*	Documentation
scm/*	Source code for the Scheme interpreter
scm/hobbit.scm	Source code for the Hobbit Scheme to C compiler
slib/*	The Scheme library distributed with Scheme
src/*	The WSL and Scheme source code for FermaT
src/adt/*	WSL source for the Abstract Data Type (Meta-WSL extensions to WSL for manipulating syntax trees)
src/scheme/*	Scheme support files (not implemented in WSL)
src/trans/*	WSL source for all the transformations
src/wslib/*	WSL source for the WSL parser, pretty printer, wsl2scheme, and other support functions
test/*	WSL test files (testing features of WSL)
test/trans/*	Transformation test files

Section 2: The Wide Spectrum Language

2.0: Note on case sensitivity

The WSL compiler is case sensitive. The reserved words of WSL are all in CAPITALS, and must be typed in capitals or the compiler won't recognize them.

Similarly, a variable, function or procedure called heLLo would be treated as distinct from a variable, function or procedure called hEllo.

2.1: Data types

WSL supports the following data types:

Integers:	3, 17, 0, -5, etc.
Boolean values:	**TRUE** or **FALSE**
Lists:	< 3, 2, 1, 2 > , < > , < "hello", 5, <1, 2 ,3> > , etc.
Sets:	<1 , 2 , 3> , < > , <2 , 3 , 5 , 7 , 11 , 13> , etc.
Strings:	"hello world", "goodbye cruel world", "", "spong", etc.

WSL also allows the creation of arrays and hash tables. (See section 2.3 below).

WSL also has a type Name for use in meta-WSL --- this is the type of variable and procedure names when they appear in a WSL program. We shall discuss this in the section on meta-WSL.

You will note from the above that there is no requirement that list elements be of the same type.

A set in WSL is internally represented as an ordered list: operations on a set are carried out under the assumption that it is indeed ordered.

This representation should not be relied upon. A list with more than one element should *always* be converted to a set via the @Make_Set function before any set operations are applied. For example:

```
x := @Make_Set(<1, 2, 3, 4>);
y := x \/ <5>;
IF n IN x THEN ... FI;
FOR elt IN x DO ... OD;
```

The IN operator can be applied to a set or a list. FOR elt IN list DO ... OD can be used with bot lists and sets. The order in which the elements of a set are processed is implementation dependent and should not be relied upon.]]

WSL supplies some Boolean-valued functions which allow you to find the type of a piece of data:

```
NUMBER?(x)    : true if x is a number
STRING?(x)    : true if x is a string
SEQUENCE?(x)  : true if x is a list or a set
```

2.2: Variables

Variables may be declared using the VAR ... ENDVAR structure.

```
VAR < v1 := a1, v2 := a2, ... >:
piece of code using the variables v1, v2, ...
ENDVAR
```

Note that the block of code must not end with a semicolon (see section 2.5).

The values a1, a2 etc. may be constants or expressions.

Example:

```
VAR < x:=4, y:="foo"++"bar" >:
PRINT(x); PRINT(y)
ENDVAR
```

will print:

```
4
foobar
```

Variables may be of any type listed in the section above WITH THE EXCEPTION of the Boolean type. Only logical expressions can be of Boolean type. WSL does NOT support Boolean-valued variables. Obviously, anything you might wish to do with Boolean-valued variables can be just as well simulated using another data type.

The **VAR** ... **ENDVAR** structures can of course be nested to any depth:

```
VAR < v1 := a1, v2 := a2, ... >:
piece of code using the variables v1, v2, ... ;
VAR < w1 := b1, w2 := b2, ... >:
piece of code using the variables w1, w2,... and also v1, v2, ...
ENDVAR;
more code using the variables v1, v2, ...
ENDVAR
```

If a variable name is reused within a nested VAR...ENDVAR, a new local variable will be created for use within that block of code. An example will make my meaning clearer:

```
VAR <a:=1>:
PRINT(a);
VAR <a:=2>:
PRINT(a)
ENDVAR;
PRINT(a)
ENDVAR
```

will print:

```
1
2
1
```

To carry over the value from the outer VAR ... ENDVAR to the inner, we can assign the new variable to have the value of the old, as follows:

```
VAR <a:="foo">:
PRINT(a);
VAR <a:=a>:
PRINT(a)
ENDVAR
ENDVAR
```

and this program will print:

```
foo
foo
```

If we do this, then the changes made to the new local variable within the inner VAR...ENDVAR do not affect the value of the original variable in the outer VAR...ENDVAR. For example:

```
VAR <a:="foo">:
PRINT(a);
VAR <a:=a>:
PRINT(a);
a:="spong";
PRINT(a)
ENDVAR;
PRINT(a)
ENDVAR
```

will print:

```
foo
foo
spong
foo
```

Note that in WSL a data type is a property of a value not of a variable. Any variable can hold a value of any type. Hence a program such as the following:

```
VAR <a:="foo">:
PRINT(a);
a:=5;
PRINT(a)
ENDVAR
```

will compile and run without any problems.

WSL does not explicitly support constants, but you are of course welcome to introduce a variable and then not change its value.

There are various "constants" supplied by WSL itself for your convenience. There is nothing to prevent you from changing any of these, but you probably don't want to.

2.3: Arrays and hash tables

Arrays and lists are two implementations (in FermaT) of WSL sequences. The same notation can be used to access and update arrays and lists, with some differences in efficiency, and some implementation restrictions.

The function ARRAY(size, init) returns an array where size is the size of the array and init is the value to which each element of the array should be initialized. For example:

```
Ar:=ARRAY(10,5)
```

sets the variable Ar to be an array with ten elements each of which is initialized to 5. The elements of the array are then Ar[1], Ar[2], ... Ar[10]. Note that the counting begins at Ar[1] --- by contrast with the language C, there is no element Ar[0].

Since the variable to which we assign the array must in any case be declared somewhere, such initializations may as well be done at the time when the variable is declared:

```
VAR <Ar:=ARRAY(10,5)>:
        code involving the array Ar
ENDVAR
```

Lists and arrays use the same operators, but are implemented differently. An assignment to a whole array will copy a pointer to the array rather than the whole array and therefore create an alias. Technically, this is an error in the implementation (the WSL specification does not allow aliasing) which might be fixed in a future version, so it should not be relied upon.

There is absolutely no requirement that the elements of an array should be of the same type.

In a similar manner, a hash table may be created by setting a variable equal to HASH_TABLE. For example:

```
T:=HASH_TABLE;
T.("foo"):=5;
T.(67):="spong";
PRINT(T.("foo"));
PRINT(T.(67))
```

will print:

```
5
spong
```

Note again that there is no restriction on the types either of the indices or the elements of the hash table.

Acessing arrays and lists will have different time efficiency. Arrays have efficient random access and update. Lists have efficient access/update for the first few elements, but accessing elements or updating elements near the end of a long list is inefficient. Lists can be copied efficiently without breaking WSL semantics, e.g.:

```
x := <1, 2, 3, 4>;
y := x;
```

The array assignment is efficient, but at the expense of breaking WSL semantics.

2.4: Expressions

WSL contains a number of built-in operators and functions. You will notice that Boolean-valued functions all have names ending in a question-mark.

Integer operators : +, -. *, /, **, **MOD**, **DIV**

The operators are listed here in ascending order of precedence. Round brackets ( ) may be used as usual to group expressions.

x**y means "x to the power y".

Integer functions : **ABS**(), **FRAC**(), **INT**(), **MAX**(), **MIN**(), **SGN**()

ABS(x) : returns the absolute value of x
FRAC(x) : returns the fractional part of a number, and so returns 0 when applied to any integer.
INT(x) : returns the integer part of x --- which is not spectacularly useful at present either.
MAX(x1,x2,...xn) : returns the maximum of x1 ... xn
MIN(x1,x2,...xn) : returns the maximum of x1 ... xn
SGN(x) : returns -1 if x < 0, 0 if x = 0 and +1 if x > 0.

Operators Z * Z -> < **TRUE**, **FALSE** > : < , > , = , <> , <= , >=

These are the usual relations on the integers.

It is easy to confuse the symbols < and > with the brackets used to specify lists and sets. The empty set must be written as < > and not as <>, or the parser will translate it as "is not equal to". Similarly, if we write the Boolean-valued expression <x,y>=<1,2>, the parser will interpret >= as "is greater than or equal to" with undesirable results. The expression would be correctly written as <x,y> = <1,2>.

Functions Z -> < **TRUE** , **FALSE** > : **ODD?**(n), **EVEN?**(n)

A pair of Boolean valued functions which return, respectively, the truth value of "n is odd" and "n is even".

Logical operators and functions: **OR** , **AND** , **NOT** , **IMPLIES?**

These are the standard logical operators. OR , AND and NOT are listed in increasing order of precedence. IMPLIES? is implemented as an ordinary function with two parameters, i.e. IMPLIES?(a, b), and so the question of precedence does not arise. IMPLIES?(a, b) is equivalent to NOT a OR b.

List/array operators and functions

a1 ++ a2 : returns the concatenation of two lists.
L[n] : returns the nth element of an array or list
L[n..m] : returns the sublist of elements n to m inclusive.
L[n..] : returns the sublist from element n to the end of the list.
BUTLAST(L) : returns the list L with the last element removed.
HEAD(L) : returns the first element of the list L[1]
IN : returns TRUE if x is an element of y.
LENGTH(L) : returns the length of the list.
LAST(L) : returns the last element of the list L[LENGTH(L)]
MAP("f",L) : returns the list obtained by applying the function f to each element of L
REDUCE("o",L) : returns the value of the expression obtained by inserting the operator o between each pair of elements of the list (which must be non-empty). For example: REDUCE("+", list) returns the sum of the values in the list.
REVERSE(L) : returns the list reversed
TAIL(L) : returns the list with the first element removed.

It is much more efficient to access and update the head of a list, rather than its tail. So if you want a list to be in the order in which elements are added to it, it's quickest to append the elements to the front and then reverse the result:

```
list := < >;
FOR i := 1 TO 10000 STEP 1 DO
list := <i> ++ list OD;
list := REVERSE(list)
```

will be much faster than:

```
list := < >;
FOR i := 1 TO 10000 STEP 1  DO
list := list ++ <i> OD
```

since the latter has to rebuild the whole list for each iteration, so it is O(n^2) in processing time. (On my machine the first takes about 30ms while the second takes nearly 11 seconds.)

Note the following identities :

L = <**HEAD**(L)> ++ **TAIL**(L)
L = **BUTLAST**(L) ++ <**LAST**(L)>
L[1] = **HEAD**(L)
L[2..] = **TAIL**(L)
LAST(L) = L[**LENGTH**(L)]
BUTLAST(L) = L[1..**LENGTH**(L)-1]

Set operators and functions

\ , \/ , /\

These are the operators of set difference, union, and intersection respectively. As usual, they are listed in increasing order of precedence. We also have:

EMPTY?(s)	: Returns true if and only if s = < >
IN	: e IN s is true if and only if e is an element of s
NOTIN	: e NOTIN s means the same as NOT(e IN s)
POWERSET(s)	: returns the set of all subsets of s
SUBSET?(s1, s2)	: true if the set s1 is a subset of s2

Functions and operators on strings

Note that WSL differs from many other languages in that the first character of each string is character 0, the second is character 1, etc.

s1 ++ s2	: returns the concatenation of two strings.
INDEX(s1, s2)	: returns the position of the first occurrence of string s1 in s2, or -1 if the string s1 is not present in s2.
INDEX(s1, s2, n)	: returns the position of the first occurrence of s1 in s2 starting at character n in s2, or -1 if there is no such occurrence. For example INDEX("456", "1234567812345678", 3) returns 3, while INDEX("456", "1234567812345678", 4) returns 11.
SLENGTH(str)	: returns the length of string str.
SUBSTR(str,i,n)	: returns n characters of str starting with the character at position i; e.g. SUBSTR("abcdef", 2, 3) returns "cde".
SUBSTR(str, I)	: returns the characters from i to the end of the string.

Note that the same operator is used for string concatenation as for list concatenation.

Because string constants are enclosed in quotes "like this", it isn't possible to directly refer to strings with quotation marks in them. A command of the form

```
PRINT("Have you read "Watership Down"?")
```

would obviously bewilder the interpreter. For this reason WSL provides a variable called Quote which contains a " character. Hence we can write:

```
PRINT("Have you read "++Quote++"Watership Down"++Quote++"?")
```

which will print

```
Have you read "Watership Down"?
```

as required.

2.5: The general structure of a WSL program

A WSL program consists of a sequence of imperative commands concatenated by semicolons. Note that the semicolons are considered to be statement separators and not statement terminators: hence, a block of code should not end with a semicolon.

As has been remarked, a program should be saved as a plain text file (conventionally with a .wsl suffix) in order for the compiler to interpret it. The last line of the program should not terminate with a semicolon (for the reason given above) but MUST terminate with a newline character.

Example:

```
PRINT("This is an example program");
PRINT("Hello world")
```

Besides imperative commands, the user may insert comments into the program using the COMMENT:"..." command or the abbreviation C:"..."

Example:

```
COMMENT:"This is a comment. It has no effect on the program";
C:"This is another comment."
```

Note that a comment is in other respects like a command in the program: it must be divided off from commands before and after it by semicolons just like a regular command. It is not an interpolation in the same way that for example curly brackets are in Pascal. Note also that you cannot have a quotes character " inside a comment, as this will give the compiler the impression that this is the end of the comment.

WSL also of course supports control structures --- conditionals and loop commands. Simple imperative commands will be discussed in section 2.6 and conditional structures and loop structures will be discussed in section 2.7 below.

In the next few sections that follow, and indeed in the review of built-in operators above, we are only looking at the reserved words of WSL --- those which are written in CAPITALS. Besides these commands and functions, WSL also has a large array of utility functions which are written in WSL and included as part of the distribution. These include procedures for allowing the user to input text, functions for generating random numbers, various list and string handling functions, and file-handling procedures. File-handling procedures will be dealt with in section 2.11; the other utility functions will be reviewed in section 2.10.

2.6: Simple imperative commands

Assignment of a value to a variable --- as you may by now have gathered --- is performed by a command of the form

variable_name:=expression

This sets the variable called variable_name to the value of the expression --- if the expression is in a correct form and can be evaluated.

<x1 := e1, x2 := e2>

is a parallel assignment which evaluates all of the expressions and then assigns to all of the variables. For example:

<x := y, y := x>

will exchange the values of the two variables.

Besides the assignment commands, the other simple commands in WSL are as follows:

{condition}	Checks whether the Boolean-valued condition is TRUE. If it's false, the program aborts.
ABORT	Aborts the program.
C:"..."	A comment in the program
COMMENT:"..."	A longer way of inserting a comment
ERROR(e1,e2,..)	Raises an error and prints the strings e1, e2, ...
POP(v,list)	Short for v:=HEAD(list); list:=TAIL(list)
PRINFLUSH(e1,e2,..)	Prints (the values of) the expressions e1,e2,...
PRINT(e1,e2,..)	Prints (the values of) the expressions e1, e2, ... on separate lines.
PUSH(list,i)	Short for list:=<i>++list
SKIP	Does nothing at all. Useful in program rewriting.

There are a few subtleties to PRINT and PRINFLUSH.

Just as we cannot set a variable equal to a Boolean value, so we cannot PRINT a Boolean value (as we would be giving PRINT a Boolean parameter). A command of the form PRINT(x=5) is syntactically invalid:

In short, the only use for such conditions is in conditional statements and loops, which we deal with in the section on control structures below.

A list or set will not PRINT out in the quite the same format as it appears in the program. For example:

PRINT(<1,"foo",<3,2,1>>)

will give the output

(1 foo (3 2 1))

As "" is the empty string, we can obtain a newline by PRINT("")

Recall that a " character is stored in the variable Quote.

2.7: Conditional structures and loops

WSL has the usual conditional structures

IF condition **THEN** block of code **FI**

and

IF condition **THEN** one block of code **ELSE** another block of code **FI**

which are interpreted as you'd expect.

Example:

IF 2+2=4 **THEN PRINT**("Maths still works"); **PRINT**("What a relief") **FI**;
IF 2+2=5 **THEN PRINT**("Reality has crashed") **ELSE PRINT**("Whew") **FI**

There is also a keyword ELSIF which can be used to make certain structures neater: things of the form

IF condition **THEN** do this **ELSE IF** another condition **THEN** do that **FI FI**

can be better written as

IF condition **THEN** do this **ELSIF** another condition **THEN** do that **FI**

By repeated use of **ELSIF** we have the equivalent in WSL of a case statement:

IF condition 1 **THEN** do thing number 1
ELSIF condition 2 **THEN** perform action do thing number 2
. . .
ELSIF condition n **THEN** perform action n
ELSE do whatever it is we do when none of conditions 1 to n is met **FI**

WSL also implements Dijkstra's guarded conditional:

D_IF condition 1 -> statement 1
[] condition 2 -> statement 2
...
[] condition n -> statement n FI

If some of the conditions 1...n is true, then this will pick one of these --- call it condition k --- and will perform action k. If none of the conditions are met, then it will abort the program.

WSL also implements standard loop structures.

DO some code **OD**

is a loop which can only be terminated by executing a statement of the form EXIT(n) where n is an integer (not a variable or expression). EXIT(n) will terminate n enclosing loops. An EXIT(n) within fewer than n loops can only appear inside a IF, D_IF or DO...OD statements (so, for example, you cannot terminate a WHILE loop via an EXIT statement).

WHILE condition **DO** ... **OD**

is the usual while loop.

FOR var := start **TO** end **STEP** step **DO** ... **OD**

is the usual FOR loop. Note that var is a local variable whose scope extends to the body of the loop only. Note also that specifying the step size is mandatory.

FOR var **IN** list **DO** ... **OD**

is an iteration over the elements of a list, array or set. The list can be an expression, e.g.

FOR comp **IN** some_list ++ some_other_list **DO** ... **OD**

The expression will be evaluated once before the loop starts. For example:

```
x := <1>;
FOR y IN x DO
x := x ++ <y> OD
```

will terminate after the first iteration.

Note that the blocks of code inside **DO** ... **OD** , **IF** ... **FI** etc. do not and should not end with a semicolon, just as the program itself should not end with a semicolon. The semicolon is not the terminator of a line, but a concatenation operator.

WSL also implements Dijksta's guarded loop:

```
D_DO condition 1 -> statement 1
[] condition 2 -> statement 2
 ...
[] condition n -> statement n OD
```

This is equivalent to:

```
WHILE cond1 OR cond2 OR ... OR condn DO
D_IF condition 1 -> statement 1
[] condition 2 -> statement 2
...
[] condition n -> statement n FI OD
```

2.8: Functions and procedures

Local procedures and functions can be defined in a WHERE clause:

```
BEGIN
...statements...
WHERE
        PROC foo(x VAR y) ==
                ... statements ... END
                FUNCT bar1(x, y) ==
                VAR < v1 := e1, v2 := e2 >:
                (expression) END
        FUNCT bar2(x, y) == :
                (expression) END
                BFUNCT spong?(x, y) ==
                VAR < v1 := e1, v2 := e2 >:
                (condition) END
END
```

A WHERE clause is just an ordinary statement which may be part of another statement or WHERE clause. The body of the WHERE clause consists of a statement sequence which may contain calls to any of the procedures or functions. Any procedure or function body may also contain calls to the other procedures or functions.

In the example procedure foo given above, the Parameter x of foo is a value parameter, while y is a value-result parameter (the final value of y is copied back into the corresponding actual parameter in the call). The

general format of a procedure call is: name(e1, e2, ... VAR v1, v2, ...) and the VAR keyword is mandatory, even if there are no value parameters.

A FUNCT cannot be Boolean valued. A BFUNCT is always Boolean valued (the B stands for Boolean). The name of a BFUNCT must terminate in a question-mark.

The body of a FUNCT or BFUNCT consists of:

1. An optional VAR < ... > which assigns to local variables. The local variables may be referenced in the final expression or condition. Note that there is no ENDVAR since this is not a VAR clause. (A PROC may introduce local variables by using a VAR ... ENDVAR clause).
2. A mandatory ":"
3. An expression or condition enclosed in parentheses.
4. The keyword END or a full stop.

A "minimal" function which contains no local variables may be defined like this:

```
FUNCT fib(n) == :
        (IF n <= 1 THEN 1 ELSE fib(n-1) + fib(n-2) FI) END
```

You will notice that a FUNCT or BFUNCT has no main body of code (by contrast with other imperative languages). The FermaT transformation system assumes that all functions (and Boolean functions) are "pure" functions which always terminate and which depend only on their parameters.

WSL also allows reference to external procedures, functions and Boolean functions --- "external" in that they are implemented in Scheme rather than WSL.

```
!XP procedure_name(parameter1,parameter2,...)
```

calls an "external procedure", i.e. a procedure written in Scheme. Similarly,

```
!XF function_name(parameter1,parameter2,...)
```

calls an external function, and

```
!XC condition_name?(parameter1,parameter2,...)
```

calls an external condition.

2.9: Action systems

A piece of code of the form

ACTIONS start:
start == some block of code **END**
name1 == another block of code **END**
name2 == yet another block of code **END**
...
...
ENDACTIONS

is an "action system", a collection of mutually recursive parameterless procedures. The system starts by executing the body of the start action (the action named on the ACTIONS line). Within the action system a statement of the form CALL action_name will call the action named action_name.

The special action call CALL Z will terminate the whole action system immediately. Z should therefore not be used as the name of an action.

An action system in which execution of every action body leads to a CALL is called a "regular action system" and is equivalent to a collection of labels and GOTOs since no action call can ever return. Note: action calls cannot appear within loop structures.

The purpose of action systems is to facilitate the translation of machine-code and spaghetti code into nice well-structured WSL. It is unlikely that you will ever want to use an action system for programming.

2.10: Utility functions

Here is a list of the utility functions which are built into WSL. As they are written in WSL as MW_FUNCTs , MW_BFUNCTs and MW_PROCs (see section 3.0 below) the name of each function necessarily begins with @. By convention, each word in the name of a utility function begins with a capital letter, and the words in the name are divided by the underline character _

@Ends_With?(str, extn): checks if the string ends with the given extn.

@Join(str, list): join a list of strings using str as the "glue".

@Join_Removing_Dups(str, list): like @Join but removes duplicates in the list first.

@List_To_String(list): converts a list to a string with spaces as separators (roughly like PRINT does).

@Make_Name(string) : convert a string to a "name"

@Make_Set(list) : converts a list into a set.

@N_String(name) : convert a "name" to a string

@Prefix?(list1,list2) : checks if list1 is a prefix of list2.

@Random(n): returns a random integer between 1 and n inclusive.

@Read_Line(port_v) : a function returning a string read from the input port assigned to the variable port_v.

@Seed_Random_State(string) : initializes the random number generator in a way that depends on the string parameter. If @Seed_Random_State is not called, then the random number generator will be initialised randomly provided the device file /dev/urandom is available on your system.

@Sort_List(list): sorts a list of names or lists of names alphabetically.

@Sort_Merge(list1,list2) : merges two sorted lists.

@Split(str): splits a string into a list of words.

@Starts_With?(str, prefix): checks if the string starts with the given prefix

@String: converts a string, number or character to a string.

@String_To_Num : converts a string to a numerical value.

@Word_In_String?(word, string): checks if the given word is in the string (treated as a space-separated list of words).

@WP(P, R) : takes a WSL program P and condition R and returns a WSL condition equivalent to the weakest precondition of P on R. Note that @WP is not defined for programs which contain loops of recursion since the weakest precondition for such programs is an infinitary formula!

2.11: File handling

In the following, "filename" can be a string or an expression returning a string, and "port" is a variable.

@Close_Input_Port(port) : closes a port returned by @Open_Input_File

@Close_Output_Port(port) : closes a port returned by @Open_Output_File

@Delete_File(filename) : deletes the given file.

@EOF?(obj) : a Boolean function which tests if the given object (returned by @Read_Char, @Peek_Char or @Read_Line) is an end of file object

@EOL?(obj) : a Boolean function which tests if the given object (returned by @Read_Char or @Peek_Char) is an end of line character.

@File_Exists?(filename) : a Boolean function which tests if the file already exists.

@Open_Input_File(filename) : a function which returns a "port" which you can use to read from the file. The filename is a string.

@Open_Output_File(filename) : a function which returns a port which you can use to write to the file.

@Peek_Char(port) : a function returning the next character which is about to be read from the port.

@Read_Char(port) : a function which returns a character read from the specified port. Returns an end of file object on end of file.

@Read_Line(port) : a function which reads a line from the file, returns either a string or an end of file object. The string doesn't include the newline character at the end (ie a blank line will return the empty string).

@Write(str, port) : writes the string str without adding a newline.

@Write_Line(str, port) : writes the line in str to the file, adding a newline character.

The variables Standard_Input_Port and Standard_Output_Port can be used to access standard input and standard output.

Simple file writing:

For the following functions, the "current output" starts out as standard output.

@Write_To(filename) : opens the given file and start writing to it. If the filename is the empty string, start writing to standard output.
@WS(str) : writes the string str to the current output
@WL(str) : writes the string str plus a newline to the current output.
@WN(num) : writes the number num to the current output
@End_Write() : closes the currently open file (if any) and redirects output back to the previous output.

@Write_To()...@End_Write operations can be nested to any depth.

2.12: Error reports and debugging

When you run a program

```
wsl program.wsl
```

then the WSL program is compiled into a temporary Scheme program, which is then run. A number of errors are not detected during the translation process and therefore are first brought to the user's attention as errors in the Scheme program which is created from the WSL code.

For example, if you make a mistake along these lines:

```
VAR <spong:=0>:
        PRINT(sping)
ENDVAR
```

then you will get an error message from Scheme, which will report the line number IN THE TEMPORARY SCHEME PROGRAM of the failure point, and the main body of which will look like this:

```
;ERROR: unbound variable:  /sping
; in expression: (display /sping)
; in scope:
;   /spong
```

As in this case, such error reports often allow you quickly to find the corresponding error in your WSL program --- especially if you have a basic knowledge of Scheme.

If the program is long and complicated, it may be useful in such cases to produce a non-temporary version of the scheme program

```
wsl2scm program.wsl program.scm
```

so as to be able to figure out which line of the Scheme program represents which line of your WSL program.

Also, the translator from WSL to Scheme can itself detect a number of syntax errors. For example, let us suppose I make a mistake of this kind:

```
VAR <x:=0> :
WHILE x<=10 DO
        IF EVEN?(x) THEN
        PRINFLUSH(x," is even; ");
        x:=x+1 OD FI
ENDVAR
```

then I should get an error message like this:

```
!!!! Line 4: Syntax Error: Duff guard syntax or un-terminated IF: 29
!!!! Line 6: Syntax Error: Missing `ENDVAR': 26
!!!! Line 7: Syntax Error: Extra characters at end of program: 999
```

Better explanation: The parser reads up to the "OD" and detects that the IF has not been closed. So it inserts a closing FI before the OD. It then reads the FI and recognises that the VAR has not been closed, so it inserts a closing ENDVAR. At this point it is expecting either a semicolon followed by another statement, or the end of the file. So the remaining "FI ENDVAR" tokens are listed as extra characters at the end of the program. So ENDVAR is not actually missing from the program above: but the syntax has already gotten so tangled that the parser doesn't know where to find it. The first syntax error, on the other hand, is telling me roughly what I need to know.

It is still possible that problems may lurk in WSL itself. If you should find such a bug in WSL, please notify its creator, including in your e-mail an example program exhibiting the bug.

Section 3: Meta-WSL

3.0: MW_BFUNCT, MW_FUNCT and MW_PROC

The "MW" procedures and functions are used for implementing the FermaT transformation system. These are statements which define procedures and functions and Boolean functions, and the functions are allowed to have side-effects (but the transformation system will still assume that there are none: so it is up to the programmer to ensure that any side-effects are "benign").

```
MW_PROC @foo(x VAR y) ==
...statements... END;

MW_FUNCT @bar(x, y) ==
VAR < v1 := e1, v2 := e2 >:
        ...statements...;
(expression) END;

MW_BFUNCT @spong?(x, y) ==
VAR < v1 := e1, v2 := e2 >:
        ...statements...;
(condition) END;

...more statements and declarations...
```

Note that the semicolon after each END is not part of the declaration, it is the normal statement separator.

Each MW_PROC, MW_FUNCT or MW_BFUNCT name must start with an "@" and each MW_BFUNCT name must end with a "?".

The body of an MW_FUNCT or MW_BFUNCT consists of:

1. An optional VAR < ... > which assigns to local variables. The local variables may be referenced in the final expression or condition. Note that there is no ENDVAR since this is not a VAR clause. (An MW_PROC may introduce local variables by using a VAR ... ENDVAR clause).
2. A mandatory ":"
3. A NON-EMPTY statement sequence
4. A mandatory ";"
5. An expression or condition enclosed in parentheses
6. The keyword END (or a full stop).

A "minimal" function which contains no local variables may be defined like this:

```
MW_FUNCT @fib(n) == : SKIP;
        (IF n <= 1 THEN 1 ELSE @fib(n-1) + @fib(n-2) FI) END
```

The SKIP statement is there to satisfy (3) above.

3.1: The internal representation of a WSL program

A WSL program is represented within WSL as a list of lists of lists ... of lists of elements. Note that this is one way of implementing a tree structure.

To be precise, a program item consists of a list of which the first element is a database table, which is used internally by FermaT; the second item is an integer giving the type of the item, and the successive elements of which (if there are any) are items which are components of the item. Note the recursive nature of this definition.

For example, the program

```
PRINT("Hello world");
PRINT("Goodbye world")
```

is, in my present implementation, internally represented by a list which would PRINT as

```
(() 17 (() 156 (() 10 (() 206 . Hello world))) (() 156 (() 10 (()
206 . Goodbye world))))
```

There is no guarantee whatsoever that the integers used to represent various features (e.g. 156 for PRINT) will be stable from version to version. Hence this representation should not be relied upon: WSL items should be created and manipulated via the appropriate MetaWSL functions and procedures.

3.2: How meta-WSL looks at a WSL program

WSL supplies a suite of constants, functions and procedures which are especially designed to handle lists which represent WSL programs. This collection of procedures and functions is known as meta-WSL.

The functions and procedures of meta-WSL are all WRITTEN IN WSL. (The source code is open to your inspection). They are all implemented as MW_PROCs, MW_FUNCTs or MW_BFUNCTs. Hence by necessity they all begin with the symbol @, and by convention the first letter in each word of the name of the procedure or function begins with a capital letter, the words being separated by the underline character. The conditional functions of course have names ending in ? , as this is compulsory.

To meta-WSL, an item consists of a specific type and either

1. a value (a constant integer, string, list etc.)
2. a list (possibly empty) of items which are components of I. Note the recursive nature of this definition.

For example, the little program above has specific type T_Statements (meaning that it is a list of statements) and two components, of which the first is returned by the function

```
@Make(T_Print, < >, <@Make(T_Expressions, < >, <@Make(T_String,
"Hello World", < >)>)>)
```

In my present implementation, T_Print is just a constant containing 156, T_Expressions is a constant containing 10, and T_String is a constant containing 206, and indeed for this specific implementation, the function

```
@Make(156, < >, <@Make(10, < >, <@Make(206,"Hello World", < >)>)>)
```

would return just the same result. Let us emphasise again : the meanings we attach to the integers may change from implementation to implementation. However, the meanings of the constant names will remain stable. The command PRINT may come to be represented by the integer 1001, in which case the constant T_Print will be defined as 1001, so that the *meaning* of T_Print remains stable between implementations.

Let's look at the program given by

@Make(T_Print, < >, <@Make(T_Expressions, < >, <@Make(T_String, “Hello World", < >)>)>)

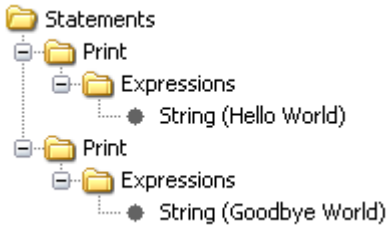
in more detail. It has specific type T_Print (meaning that it is a PRINT statement) and one component. This component is an item of specific type T_Expressions, which means that it is a LIST of expressions --- remember that the syntax of the PRINT command is PRINT(e1,e2,...,en). This list, in this case, consists of just one item of specific type T_String (meaning that it is a string) and a value --- the string "Hello world".

Such constants USUALLY but not invariably have the name that you would expect: the specific type of an D_IF statement is T_D_If; the specific type of an ABORT statement is T_Abort; the specific type of x<>10 is T_Not_Equal. However, there are exceptions: the specific type of an IF statement is T_Cond. T_If is the specific type of a conditional expression.

A complete list of such constants may be found by inspecting the file fermat3/src/adt/WSL-init.wsl

The components of an item invariably occur in exactly the order that you would expect: the same order as they appear when you program WSL. So, for example a FOR loop will have five components: the name of the loop variable, the start value, the end value, the step size and the list of statements inside the DO...OD clause, and they will appear in that order.

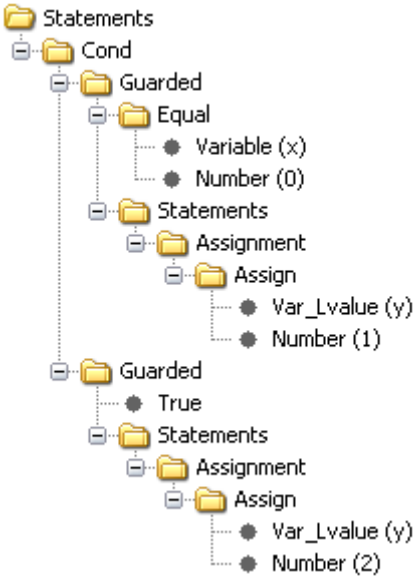
I have remarked that the program essentially has the structure of a tree, and we might conveniently represent our program visually as such.



This is simple enough. Now let's look at

IF x = 0 **THEN** y := 1 **ELSE** y := 2 **FI**

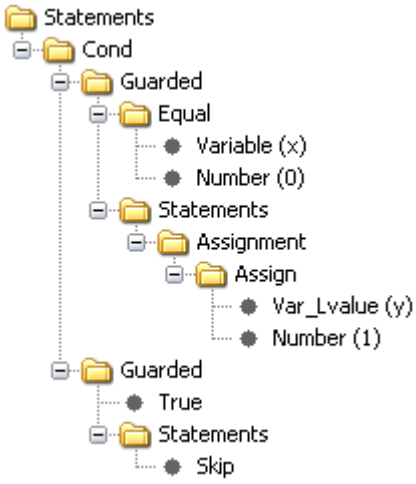
This has the following structure:



Observe carefully the way that IF...THEN...ELSE...FI is represented. In the same way, the program

IF x = 0 **THEN** y:=1 **FI**

would be represented like this:



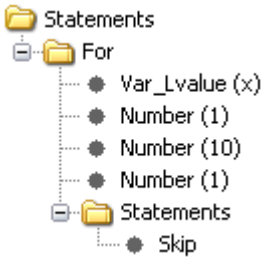
The purpose of all this is to give conditional statements a nice uniform structure: every conditional statement has as its components a list of items of type T_Guarded.

Observe also that (the numbers identifying) variables have the type T_Variable when we inspect their value but T_Var_Lvalue when they stand on the left hand side of an assignment.

As a final example, let's look at a loop structure.

FOR x:=1 **TO** 10 **STEP** 1 **DO SKIP OD**

The syntax tree looks like this:



Note that as x is to be assigned it has type T_Var_Lvalue.

Meta-WSL provides you with functions which allow you to find whether a specific type takes a value or a list of components.

@Has_Value_Type?(st): true if the specific type st is one which has a value
@Has_Comps_Type?(st): true if the specific type st is one which may have components

3.3: General types

We may also consider the "general type" of an item --- as distinct from its specific type.

A statement has general type T_Statement.
An expression has general type T_Expression.
A condition has general type T_Condition.
A constant (integers, strings, etc) has general type T_Value.
The definition of a FUNCT, PROC etc has general type T_Definition.
A list of statements has general type T_Statements
A list of expressions has general type T_Expression
A list of conditions has general type T_Conditions

These last three may also be considered as specific types. That is to say, a list of statements has general type T_Statements and also specific type T_Statements.

3.4: Specific types and components of WSL commands

Here is a summary of the specific types and the components of structures found in WSL as described in section 2 above.

The types given for the components may be specific or general types, depending on the structure of the item.

When describing the types of an items components below, I shall use the notation 2*T_Type, 3*T_Type etc to mean a list of 2, 3 etc components of type T_Type. I shall write n*T_Type to indicate that any number of components of type T_Type are acceptable.

Note that T_Expressions has components n*T_Expression; T_Statements has components n*T_Statement, etc. (This rule breaks down only on T_Lvalues, which is more likely to have as components a list of items of type T_Var_Lvalue).

Item	Specific type	Components of type
Integer	T_Number	(takes a value)
List	T_Sequence	(takes a value)
Set	T_Set	(takes a value)
String	T_String	(takes a value)
+	T_Plus	n*T_Expression
-	T_Minus	n*T_Expression
*	T_Times	n*T_Expression
/	T_Divide	n*T_Expression
**	T_Exponent	2*T_Expression
=	T_Equal	2*T_Expression
<	T_Less	2*T_Expression
>	T_Greater	2*T_Expression
<=	T_Less_Eq	2*T_Expression

Item	Specific type	Components of type
>=	T_Greater_Eq	2*T_Expression
<>	T_Not_Equal	2*T_Expression
++	T_Concat	2*T_Expression
list[n]	T_Aref	T_Expression, T_Expressions
list[n..m]	T_Subseg	3*T_Expression
list[n..]	T_Final_Seg	2*T_Expression
\	T_Set_Diff	2*T_Expression
\/	T_Union	2*T_Expression
/\	T_Intersection	2*T_Expression
{..}	T_Assert	T_Condition
bfunct_name(..)	T_BFunct_Call	T_Name, T_Expressions
function_name(..)	T_Funct_Call	T_Name, T_Expressions
procedure_name(..)	T_Proc_Call	T_Name, T_Expressions, T_Lvalues
ABORT	T_Abort	
ABS	T_Abs	T_Expression
ACTIONS ... ENDACTIONS	T_A_S	T_Name,T_Actions
AND	T_And	2*T_Condition
ARRAY	T_Array	2*T_Expression
BEGIN ... WHERE ... END	T_Where	T_Statements, T_Definitions
BFUNCT	T_BFunct	T_Name, T_Lvalues, T_Assigns, T_Condition
BUTLAST	T_Butlast	T_Expression
CALL	T_Call	(takes a value)
D_IF	T_D_If	n*T_Guarded
D_DO	T_D_Do	n*T_Guarded
DIV	T_Div	2*T_Expression
DO	T_Floop	T_Statements
EMPTY?	T_Empty	T_Expression
ERROR	T_Error	T_Expressions
EVEN?	T_Even	T_Expression
FALSE	T_False	
FOR IN	T_For_In	T_Var_Lvalue, T_Expression, T_Statements
FOR TO STEP	T_For	T_Var_Lvalue, 3*T_Expression, T_Statements
FRAC	T_Frac	T_Expression
FUNCT	T_Funct	T_Name, T_Lvalues, T_Assigns, T_Expression
HASH_TABLE	T_Hash_Table	
HEAD	T_Head	T_Expression

Item	Specific type	Components of type
IF THEN	T_Cond	2*T_Guarded
IF THEN ELSE	T_Cond	2*T_Guarded
IF THEN ELSIF ...	T_Cond	n*T_Guarded
IMPLIES?	T_Implies	2*T_Condition
IN	T_In	2*T_Expression
INDEX	T_Index	2*Expression
INT	T_Int	T_Expression
LAST	T_Last	T_Expression
LENGTH	T_Length	T_Expression
MAP	T_Map	T_Name, T_Variable
MAX	T_Max	n*T_Expression
MEMBER?	T_Member	2*T_Expression
MIN	T_Min	n*T_Expression
MOD	T_Mod	2*T_Expression
NOT	T_Not	T_Condition
NOTIN	T_Not_In	2*T_Expression
NUMBER?	T_Numberq	T_Expression
ODD?	T_Odd	T_Expression
OR	T_Or	2*T_Condition
POP	T_Pop	2*T_Var_Lvalue
POWERSET	T_Powerset	T_Expression
PRINFLUSH	T_Prinflush	T_Expressions
PRINT	T_Print	T_Expressions
PROC	T_Proc	T_Name, 2*T_Lvalues, T_Statements
PUSH	T_Push	T_Var_Lvalue,T_Expression
REDUCE	T_Reduce	T_Name, T_Variable
REVERSE	T_Reverse	T_Expression
SEQUENCE?	T_Sequenceq	T_Expression
SGN	T_Sgn	T_Expression
SKIP	T_Skip	
SLENGTH	T_Slength	T_Expression
STRING?	T_Stringq	T_Expression
SUBSET?	T_Subset	T_Expression
SUBSTR	T_Substr	T_Expressions
TAIL	T_Tail	T_Expression
TRUE	T_True	
VAR	T_Var	T_Assigns, T_Statements
WHILE	T_While	T_Condition, T_Statements
!XC name(..)	T_X_BFunct_Call	T_Name,T_Expressions
!XF name(..)	T_X_Funct_Call	T_Name,T_Expressions

Item	Specific type	Components of type
!XP name(..)	T_X_Proc_Call	T_Name,T_Expressions
Also, note	T_Guarded	T_Condition, T_Statements

Note that the suffix q on T_Numberq, T_Setq and T_Stringq are there to distinguish them from T_Number, T_Set and T_String. In all other cases a final question-mark in a keyword disappears entirely in the name of its specific type: e.g. the type of EVEN? is T_Even.

3.5: Making WSL items: @Make and FILL ... ENDFILL

The function @Make(type, value, comps) constructs a new item with the given type, value and list of components.

The function @Make_Name(string) converts a string to a "name" suitable to use as the value for various item types. For example, a T_Variable needs a "name" as the value: @Make(T_Variable, @Make_Name("foo"), < >).

For example:

```
@Make(T_Assignment, < >,
<@Make(T_Assign, < >, <@Make(T_Lvalue, @Make_Name("x"), < >),
@Make(T_Number, 3, < >)>)>)
```

returns the list representing the program

```
x:=3
```

Note that the "name" is actually an index into the array N_Symbol_Table. The hash table N_String_To_Symbol returns the array index for each string in the symbol table); similarly @N_String(name) converts a name value to a string (this is implemented via the N_String_To_Symbol lookup table).

To save you trouble, WSL provides a structure

```
FILL general_type item_schema ENDFILL
```

which returns a program item of the given general type as specified by the text program given as the second parameter. For example:

```
FILL Statement PRINT("Hello world") ENDFILL
```

returns the list representing the program

```
PRINT("Hello world")
```

while

```
FILL Statements
PRINT("Hello world");
PRINT("Goodbye world")
ENDFILL
```

returns the list representing

```
PRINT("Hello world");
PRINT("Goodbye world")
```

as you would hope.

The item schema can refer to variables containing items. For example:

```
n := @Make(T_Number, 3, < >);

S := FILL Statement x := ~?n ENDFILL
```

will leave S containing a list representing the program

```
x:=3
```

Note that if we had accidentally written

```
n := @Make(T_Number, 3, < >);
S := FILL Statement x := n ENDFILL
```

then S would end up containing a list representing

```
x:=n
```

The patterns in a schema take the form: "~" plus "+", "*" or "?" plus a variable name. For example: ~?S1, ~+vars, ~*B2

"~?foo" means insert the single item contained in variable foo.
"~+foo" means splice in the non-empty list of items stored in foo.
"~*foo" means splice in the possibly-empty list of items stored in foo.

For example:

```
S1 := <FILL Statement x := x+1 ENDFILL, FILL Statement y := y+x ENDFILL>;
S2 := FILL Statements x :=1; ~*S1 ENDFILL
```

sets S2 to an item which is a statement sequence with three components, representing the program:

```
x := 1;
x := x+1;
y := y+x
```

Note that whereas

```
FILL Statement SKIP ENDFILL
```

is equivalent to

```
@Make(T_Skip, < >, < >)
```

the expression

```
FILL Statements SKIP ENDFILL
```

is equivalent to

```
@Make(T_Statements, < >, <@Make(T_Skip, < >, < >)>)
```

3.6: Meta-WSL functions on program items

I^n returns the nth component of item I. I^^list may be defined recursively: if list=< > then I^^list is I; otherwise, I^^list is (I^HEAD(list))^^TAIL(list). Alternatively, if you choose: if list=< > then I^^list is I; otherwise, I^^list is (I^BUTLAST(list))^LAST(list). To put it still another way, if list = <list[1],list[2],...,list[n]>, then I^^list is the list[n]-th component of the list[n-1]-th component of ... of the list[2]-th component of the list[1]-th component of I.

Besides these operators, we have the following functions on program items:

@Assd_Only(I) = @Assigned(I) \ @Used(I).

@Assd_To_Self(I): the set of variables which are only used in assignments to themselves.

@Assigned(I): the set of all variables assigned in I.

@Calls(I): returns a list of pairs the form <<name1, n1>, <name2, n2>,...> giving the actions called and number of times each action is called.

@Call_Freq(n, I): returns how many times the action n is called in I.

@Clobbered(I): the set of variables which are always assigned in I.

@Cs(I): the components of I (only works on items which don't have a value) @Components(I): the components of I (works on any item)

@Cs?(I), @Components?(I): true if I has components.

@Funct_Calls(I): Like @Calls, but counts FUNCT calls.

@Funct_Call_Freq(n, I) returns how many times the procedure n is called in item I.

@GT(I): the generic type of I.

@Proc_Calls(I): like @Calls, but counts PROC calls.

@Proc_Call_Freq(n, I): returns how many time procedure n is called by I.

@Redefined(I): the set of variables which are always redefined, and not in terms of themselves.

@Size(I):the number of components in I.

@ST(I): the specific type of I.

@UBA(I): the set of variables whose initial values are used before any new values are assigned to them in I.

@Used(I): the set of all variables referenced in I.

@Used_Only = @Used(I) \ @Assigned(I).

@V(I): the value of I (only works on items with a value)-

@Valid_Posn?(I, list): checks whether I^^list is defined-

@Value(I): the value of I or < > (works on any item)-

@Variables(I): the set of all variables in I-

@X_Funct_Calls(I): Like @Funct_Calls, but counts external function calls only.

@X_Funct_Call_Freq(n, I): Counts how often the external function n is called by I.

3.7: The current program and the current item

To assist in the transformation of programs, Meta-WSL has the concept of the current program and the current program item within the current program. The current program is returned by the parameterless function @Program and the current item is returned by the parameterless function @I.

Similarly the parent and grandparent of the current item are returned by @Parent and @GParent respectively. If the current item doesn't have a parent (respectively, grandparent) in the current program, then these functions will cause the program to crash: they must be used carefully.

To start with, the current program will be the empty list, and the current item will be the current program. You can set the current program to be an item P like this:

@New_Program(P).

@Program will now return P, and as the program has just been reset, @I will also return P.

NOTE THAT as @Program and @I are parameterless functions and not variables, it is NOT POSSIBLE to write @Program:=P to achieve the same effect.

The idea of the current item is that it is the particular bit of the current program we can most easily inspect and edit. The position of the current item within the program is given by a list, as explained at the start of section 3.5. Hence as the parameterless function @Posn returns the position of @I in @Program, we have @I = @Program^^@Posn.

WSL provides the following procedures and functions for moving the position of the current item.

@Down moves to the first component of the current item.

@Down? checks whether the move @Down is possible.

@Down_Last moves to the last component of the current item.

@Down_To(n) moves to the nth component of the current item.

@Find_Type(type) moves "forwards" (down and right) to the first component with the given specific type.

@Goto(posn): move to the given position.

@Left moves to the left, i.e. to the component of the parent of the current item which precedes the current item in the list of components of @Parent

@Left? checks whether the move @Left is possible.

@Posn: returns the current position.

@Posn_n: returns the current position in the current parent: so that @Posn_n = LAST(@Posn)

@Right moves to the right, i.e. to the component of the parent of the current item which succeeds the current item in the list of components of @Parent.

@To(n) moves to the nth sibling of the current item.

@Up moves up, i.e. to the parent of the current item.

@Up? checks whether the move @Up is possible.

3.8: Editing the current item

Meta-WSL provides you with the following tools for editing the current item.

@Buffer: return the item or list of items in the buffer.

@Clever_Delete: delete the current item and "fix up" the syntax of the resulting program. This may change @Posn, but the resulting position will be valid.

@Cut: delete the current item and store it in the cut buffer.

@Cut_Rest: delete any items to the right of the current item and store the list of deleted items in the cut buffer.

@Delete: delete the current item (without worrying about the resulting syntax) This leaves @Posn unchanged, even if the resulting position is invalid.

@Delete_Rest: delete any items to the right of the current item. This leaves @I and @Posn unchanged.

@Edit: start editing the current item. This will create a new program in which the current item is the whole program. This has two uses:

1. Editing operations on the current item are more efficient since they only need to create a new item, not a whole new program.
2. The result of the edit can be "undone" very efficiently.

@Edit_Parent: start editing the current item, but the parent of this item is the new current program. This is like @Edit but preserves a little more "context".

@End_Edit: stop editing the current item and paste the result back into the original program.

@Paste_Over(I): replace the current item by I @Paste_Before(I): insert I as a new sibling to the left of the current item. @Paste_After(I): insert I as a new sibling to the right of the current item.

@Rename(old, new): rename a variable throughout the current item.

@Splice_Over(L): replace the current item by the list of items in L

@Splice_Over(< >) is equivalent to @Delete.

@Splice_Before(L), @Splice_After(L): insert the list L of items to the left or right of the current item. @Splice_Before(<I>) is equivalent to @Paste_Before(I).

@Trans(n, data): execute transformation n on the current item, passing the given data to the @XXX_Code() procedure. Assumes that the transformation is valid (i.e. @Trans?(n) would be true if called). For further information see section 4.0 below.

@Trans?(n): true if transformation n is valid on the current item. This works by calling the appropriate @XXX_Test?() procedure and checking whether @Pass or @Fail was called. If the result is false (i.e. @Fail was called) then @Fail_Message returns the message passed to @Fail.

@Undo_Edit: throw away the current program and go back to the original program.

Note that @Edit ... @End_Edit/@Undo_Edit operations can be nested to any depth.

3.9: FOREACH, ATEACH and IFMATCH

The **FOREACH** structure is very useful for program rewriting:

FOREACH type **DO** ...statements... **OD**

This iterates over every component of the current item, executing the body of the code on each component of the right type. The iteration is done in a "bottom up" fashion: i.e. all subcomponents of a component will be processed before the component itself.

The possible types are:

Statement
Statements
Terminal Statement
Terminal Statements
STS (short for Simple Terminal Statement)
NAS (short for Non-Action System)
Expression
Condition
Variable
Global Variable
Lvalue

FOREACH NAS **DO** ... **OD**

will not descend into an action system. This is used for unfolding action calls:

```
FOREACH NAS DO
        IF @ST(@I) = T_Call AND @V(@I) = name
                THEN @Splice_Over(body) FI OD
```

where we don't need to worry about a sub-action system which uses the same action name.

While executing the body of the loop, the currently selected item will appear to be the entire program (if the item is a statement then it will appear as the only statement in an outer statement sequence). The body of the loop can delete the statement or insert extra statements and the FOREACH loop will sort out the resulting syntax.

For example, this loop:

```
FOREACH Statement DO
        IF @ST(@I) = T_Skip THEN @Delete FI OD;
```

will transform:

WHILE x = 0 **DO SKIP OD**

into the assertion:

{x <> 0}

There is an alternative structure

ATEACH type **DO** ... statements ... **OD**

This is similar to a **FOREACH** loop but with three differences:

1. Components are processed in a top down fashion: i.e. an item is processed first and then the components of the (processed) item are processed.
2. Executing a @Fail() in the loop body will cause the loop to terminate.
3. A new program is not created for the current item: this means that the "context" of the item is available to the loop body, but care must be taken to return to the starting point if the loop body contains move operations. Otherwise the loop might miss out some components or iterate indefinitely. For example this WSL program will loop forever:

```
@New_Program(FILL Statements SKIP; SKIP ENDFILL);
        ATEACH Statement DO
                IF @Left? THEN @Left FI OD
```

The list of types is the same as for **FOREACH**.

The **IFMATCH**...**ENDMATCH** structure is also useful.

```
IFMATCH type schema
THEN ...statements...
ELSE ...statements... ENDMATCH
```

This statement does a pattern match on the current item. Pattern variables in the schema (e.g. ~?S) are either matched against the current value of the corresponding variable (e.g. S for the pattern variable ~?S) or, if the current value is < > then the corresponding variable is set to the matched item or list of items.

For example, here is a statement which will match against a simple IF statement and rewrite it by reversing the two arms of the IF and inverting the test:

```
VAR < B := < >, S1 := < >, S2 := < > >:
        IFMATCH Statement IF ~?B THEN ~?S1 ELSE ~?S2 FI
                THEN B := @Not(B);
                @Paste_Over(FILL Statement IF ~?B THEN ~?S2 ELSE ~?S1 FI ENDFILL)
        ENDMATCH ENDVAR
```

Whereas the following program

```
VAR < B := < >, S1 := FILL Statements SKIP ENDFILL, S2 := < > >:
        IFMATCH Statement IF ~?B THEN ~?S1 ELSE ~?S2 FI
                THEN B := @Not(B);
                @Paste_Over(FILL Statement IF ~?B THEN ~?S2 FI ENDFILL)
        ENDMATCH ENDVAR
```

matches an IF statement of the form

```
IF condition THEN SKIP ELSE statements FI
```

Note the result of setting S1 := **FILL** Statements **SKIP ENDFILL**

3.10: The maths simplifier

Meta-WSL provides you with a number of functions which automatically simplify expressions and conditions:

@And(b1, b2): constructs the condition b1 AND b2 and then tries to simplify it.

@False?(cond): checks if the given item simplifies to FALSE.

@Implies?(b1, b2): checks if the condition "NOT b1 OR b2" simplifies to TRUE.

@Invert(x, v, exp): exp should contain exactly one occurrence of the name v. This returns an expression exp2 which inverts the effect of exp such that "x := exp; x := exp2" is equivalent to SKIP. In other words, replacing v in exp by exp2 will give an expression that simplifies to x.

For example:

```
@Invert(FILL Expression x ENDFILL, @Make_Name("v"),
                FILL Expression 2*v - 1 ENDFILL)
```

will return the expression (x + 1)/2

```
@Invert(FILL Expression x ENDFILL, @Make_Name("v"),
                FILL Expression 3 - 2*v ENDFILL)
```

will return the expression (3 - x)/2

@Not(b): constructs the condition NOT(b) and then tries to simplify it.

@Or(b1,b2): constructs the condition b1 OR b2 and then tries to simplify it.

@Simplify(item, budget): return a simplified item. The integer "budget" indicates how much "effort" to exert in trying to simplify the item.

@Simplify_Expn(expn), @Simplify_Cond(condition): call @Simplify with the default budget value of 10.

@True?(cond) checks if the given condition will simplify to TRUE.

3.11: Metrics on program items

A number of functions are provided which measure the complexity of a given WSL item according to a variety of metrics.

@Stat_Types(I)	: return set of statement types appearing in I
@Total_Size(I)	: total number of nodes (items) in I
@Stat_Count(I)	: total number of statement items
@Gen_Type_Count(type, I)	: number of occurrences of given generic type
@Spec_Type_Count(type, I)	: ditto for a specific type
@McCabe(I)	: McCabe cyclometric complexity measure for I
@CFDF_Metric(I)	: control-flow / data-flow metric for I
@BL_Metric(I)	: branch-loop metric for I
@Struct_Metric(I)	: a weighted sum over all the items in I

Section 4: Transformations

4.0: Writing, installing and testing transformations

The source files for the available transformations are in the folder fermat3/src/trans.

Each transformation is written to operate on the current item @I. Hence when we call the procedure

```
@Trans(n,data).
```

it applies transformation n on @I with the given data. Most transformations do not require any data as parameters.

If you call the transformation from the command line interface:

```
dotrans FILENAME1.wsl FILENAME2.wsl Transformation
```

then before running the transformation, the current item will be set to the program in FILENAME1.wsl, so that it will transform the whole program.

Each transformation consists of a program called trans_name.wsl (where Trans_Name is the name of the transformation) containing

1. An MW_PROC without parameters called @Trans_Name_Test which raises errors if the item is not suitable for the transformation.
2. An MW_PROC called @Trans_Name_Code and taking as its parameter the data to be passed to the transformation (if any).
3. Any auxiliary functions or procedures useful to Trans_Name_Code or Trans_Name_Test.
4. A final SKIP --- which constitutes the main body of the programtransformation_name.wsl, thus ensuring that technically it is indeed a well formed WSL program.

In addition, for each transformation there is an auxiliary file transformation_name_d.wsl which registers the transformation with the system, and having the following form:

```
TR_Trans_Name := @New_TR_Number;
TRs_Name[TR_Trans_Name] := "Trans Name";
TRs_Proc_Name[TR_Trans_Name] := "Trans_Name";
TRs_Test[TR_Trans_Name]:=!XF funct(@Trans_Name_Test);
TRs_Code[TR_Trans_Name]:=!XF funct(@Trans_Name_Code);
TRs_Keywords[TR_Trans_Name] := < "key" , "words" >;
TRs_Help[TR_Trans_Name] := "This transformation does the following...";
TRs_Prompt[TR_Trans_Name] := "";
TRs_Data_Gen_Type[TR_Trans_Name] := ""
```

Hence if you have a procedure which rewrites program items, you can put it into the form given above, and put it together with an appropriate _d.wsl file into fermat3/src/trans. Then rebuild FermaT. Your procedure is now a FermaT transformation.

You will find various tests for the transformations in the folder fermat3/test/trans. Each test file has the name trans_name_TEST.wsl where as before Trans_Name is the name of the transformation.

A test file consists of a set of @Test_Trans commands. The arguments of Test_Trans consist of:

1. A string identifying the test: "nth test of `Trans Name'".
2. The item to be tested.
3. The position in this item at which the transformation should be applied (given as a list --- see section 3.6).
4. The data for the transformation: if there is none, an empty set
5. Either the code which should result from the transformation (if the item to be transformed ought to pass Trans_Name_Test in trans_name.wsl) OR the string "Fail" if it should be failed by Trans_Name_Test.

These tests are applied every time FermaT is compiled.

4.1: The transformations included with FermaT

Here is a brief summary of the transformations included with this distribution:

Absorb_Left: This transformation will absorb into the selected statement the one that precedes it.

Absorb_Right: This transformation will absorb into the selected statement the one that follows it.

Actions_To_Procs: Search for actions which call one other action and make them into procs.

Actions_To_Where: converts an action system to a WHERE clause

Add_Assertion: This transformation will add an assertion after the current item, if some suitable information can be ascertained.

Add_Left: This transformation will add the selected statement (or sequence of statements) into the statement that precedes it without doing further simplification.

Align_Nested_Statements: This transformation takes a guarded clause whose first statement is a IF and integrates it with the outer condition by absorbing the other guarded statements into the inner IF, and then modifying its conditions appropriately. This is the converse of Partially Join Cases.

Collapse_Action_System: Collapse Action System will use simplifications and substitution to transform an action system into a sequence of statements, possibly inside a DO loop.

Collapse_All_Action_Systems: Collapse All Action Systems will attempt to collapse the action systems within a program which is a WHERE structure.

Combine_Wheres: will combine two nested WHERE structures into one structure which will contain the definitions from each of the original WHERE structures. The selected WHERE structure will be merged into an enclosing one if there is one or, failing that, into an enclosed WHERE structure.

Constant_Propagation: Constant Propagation finds assignments of constants to variables in the selected item and propagates the values through the selected item (replacing variables in expressions by the appropriate values)

D_Do_To_Floop: converts a D_DO loop to an equivalent DO ... OD loop.

Delete_All_Assertions: This transformation will delete all the `ASSERT' statements within the selected code. If the resulting code is not syntactically correct, the program will be `tidied up' which may well result in the re-instatement of `ASSERT' or `SKIP' statements.

Delete_All_Comments: This transformation will delete all the `COMMENT' statements within the selected code. If the resulting code is not syntactically correct, the program will be `tidied up' which may well result in the insertion of `SKIP' statements.

Delete_All_Redundant: Delete All Redundant searches for redundant statements and deletes all the ones it finds. A statement is `Redundant' if it calls nothing external and the variables it modifies will all be assigned again before their values are accessed.

Delete_All_Skips: This transformation will delete all the `SKIP' statements within the selected code. If the resulting code is not syntactically correct, the program will be `tidied up' which may well result in the reinstatement of `SKIP' statements.

Delete_Item: This transformation will delete a program item that is redundant or unreachable

Delete_Redundant_Statement: Delete Redundant Statement checks whether the current statement is `Redundant' (because it calls nothing external and the variables it modifies will all be assigned again before their values are accessed). If so, it deletes the Statement.

Delete_Unreachable_Code: Delete Unreachable Code will remove unreachable statements in the selected object. It will also remove unreachable cases in an IF statement, e.g. those which follow a FALSE guard.

Delete_What_Follows: Delete What Follows will delete the code which follows the selected item if it can never be executed.

Double_To_Single_Loop: Double to Single Loop will convert a double nested loop to a single loop, if this can be done without significantly increasing the size of the program.

Else_If_To_Elsif: This transformation will replace an `Else' clause which contains an `If' statement with an `Elsif' clause. The transformation can be selected with either the outer `If' statement, or the `Else' clause selected.

Elsif_To_Else_If: This transformation will replace an `Elsif' clause in an `If' statement with an `Else' clause which itself contains an `If' statement. The transformation can be selected with either the `If' statement, or the `Elsif' clause selected.

Expand_And_Separate_All: Expand And Separate All will attempt to apply the transformation Expand and Separate to the first statement in each action in an action system.

Expand_And_Separate: Expand And Separate will expand the selected IF statement to include all the following statements, then separate all possible statements from the resulting IF. This is probably only useful if the IF includes a CALL, EXIT etc. which is duplicated in the following statements, otherwise it will probably achieve nothing.

Expand_Call: Expand_Call will replace a call to an action, procedure or function with the corresponding definition.

Expand_Forward: Expand_Forward will copy the following statement into the end of each branch of the selected IF or D_IF statement. It differs from Absorb Right in that the statement is only absorbed into the `top level' of the selected IF.

Find_Entry_Points: Find possible entry points in the action system by looking for actions which are not reachable from the starting action. Create a new starting action with a Dijkstra IF statement.

Find_Terminals: Find and mark the terminal statements in the selected statement. If a terminal statement is a local procedure call, apply recursively to the procedure body.

Floop_To_While: Convert a suitable DO...OD loop to a While loop.

Force_Double_To_Single_Loop: Force Double - Single Loop will convert a double nested loop to a single loop, regardless of any increase in program size which this causes.

Fully_Absorb_Right: This transformation will absorb into the selected statement all the statements that follow it.

Fully_Expand_Forward: Apply Expand Forward as often as possible.

Global_To_Pars: converts global variables to parameters.

If_To_Case: turns IF statements into case statements.

Insert_Assertion: This transformation will add an assertion inside the current item, if some suitable information can be ascertained.

Join_All_Cases: This transformation will join any guards in an `If' statement which contain the same sequence of statements (thus reducing their number) by changing the conditions of all the guards as appropriate.

Make_Basic_Block_Form: This transformation will put an action system into Basic Block Form.

Make_Proc: `Make Procedure' will make a procedure from the body of an action or from a list of statements.

Merge_Calls_In_Action: Merge Calls in Action will attempt to merge calls which call the same action, in the selected action.

Merge_Calls_In_System: Use absorption to reduce the number of calls in an action system.

Merge_Left: This transformation will merge the selected statement (or sequence of statements) into the statement that precedes it.

Merge_Right: This transformation will merge the selected statement into the statement that precedes it.

Meta_Trans: Convert a FOREACH with a long sequence of IFMATCH commands to a more efficient form.

Move_Comment_Left: Moves the selected Comment Left.

Move_Comment_Right: Moves the selected Comment Right.

Move_Comments: Move Comments will move any comments which appear at the end of actions within an action system and which follow a call. The comments will be moved in front of the call.

Move_To_Left: This transformation will move the selected item to the left so that it is exchanged with the item that precedes it.

Move_To_Right: This transformation will move the selected item to the right so that it is exchanged with the item that follows it.

Partially_Join_Cases: This transformation will join any guards in an IF statement which contain almost the same sequence of statements (thus reducing their number) by introducing a nested IF and changing the conditions of all the guards as appropriate.

Push_Pop: Replaces PUSH(stack, v); ... POP(stack, v) by VAR < v := v >: ... ENDVAR, and replaces PUSH(stack, v1); ... POP(stack, v2) by VAR < tmp := v1 >: ... v2 := tmp ENDVAR

Recursion_To_Loop: will replace the body of a recursive action if possible by an equivalent loop structure.

Reduce_Loop: Automatically make the body of a DO...OD reducible (by introducing new procedures as # necessary) and either remove the loop (if it is a dummy loop) or convert the loop to a WHILE loop (if the loop is a proper sequence).

Reduce_Multiple_Loops: This transformation will reduce the number of multiply nested loops to a minimum.

Remove_All_Redundant_Vars: Remove All Redundant Vars applies Remove Redundant Vars to every VAR structure in the statement or sequence.

Remove_All_Redundant_Vars: takes out as many local variables as possible from the selected VAR structure. If they can all be taken out, the VAR is replaced by its (possibly modified) body.

Remove_Comment: removes the selected item if it's a comment.

Remove_Dummy_Loop: Remove Dummy Loop will remove a DO loop which is redundant.

Remove_Elem_Actions: Remove Unnecessary Actions will remove any actions in the selected action system which merely call another action or are not called at all. Calls to the deleted action will be replaced by calls to the other action.

Rename_Defns: Ensures that no two procedures have the same name.

Rename_Local_Vars: Ensures that no two local variables have the same name.

Replace_Accs_With_Value: This transformation will apply Replace With Value to all variables with the names a0, a1, a2 and a3 in the selected item.

Replace_With_Value: This transformation will replace a variable (in an expression) by its value ---- provided that that value can be uniquely determined at that point in the program.

Reverse_Order: This transformation will reverse the order of most two-component items; in particular expressions, conditions and `If's which have two branches.

Separate_Both: will take code out to the right and the left of the selected structure.

Separate_Left: will take code out to the left of the selected structure. As much code as possible will be taken out; if all the statements are taken out then the original containing structure will be removed.

Separate_Right: will take code out to the right of the selected structure.

Simplify_Action_System: Simplify action system will attempt to remove actions and calls from an action system by successively applying simplifying transformations. As many of the actions as possible will be eliminated without making the program significantly larger.

Simplify: This transformation will simplify any component as fully as possible.

Simplify_If: Simplify If will remove false cases from an IF statement, and any cases whose conditions imply earlier conditions. Any repeated statements which can be taken outside the if will be, and the conditions will be simplified if possible.

Simplify_Item: This transformation will simplify an item, but not recursively simplify the components inside it. In particular, the transformation will simplify expressions, conditions and degenerate conditional, local variable and loop statements.

Substitute_And_Delete: Substitute and Delete will replace all calls to an action, procedure or function with the corresponding definition, and delete the definition

Substitute_And_Delete_List: Substitute and Delete List will replace all calls to any action within the selected list of actions with the corresponding definition and delete the definition. Actions which are called more than once will not be affected.

Substitute Once-Called Actions: replaces calls to actions which are called only once with their definitions , and then deletes the respective definitions.

Take_Out_Left: This transformation will take the selected item out of the enclosing structure towards the left.

Take_Out_Of_Loop: This transformation will take the selected item out of an appropriate enclosing loop towards the right.

Take_Out_Right: This transformation will take the selected item out of the enclosing structure towards the right.

Unfold_Proc_Call: Unfold the selected procedure call, replacing it with a copy of the procedure body.

Unfold_Proc_Calls: Unfold Proc Calls searches for procedures which are only called once, unfolds the call and removes the procedure.

Use_Assertion: if the current item is an assertion, this tries to use the assertion to simplify the following program.

While_To_Floop: changes a WHILE loop to an equivalent DO..OD loop.