

Unimal

Unified macro language

Language-independent macro processor
for programmers

Version 2.1

Documentation revision 2.1a



MacroExpressions
<http://www.macroexpressions.com>

Table of contents

0. PRELIMINARY NOTES	5
0.1. WHAT'S NEW IN UNIMAL 2.1	5
0.1.1. <i>Motivation</i>	5
0.1.2. <i>Macro definitions</i>	5
0.1.3. <i>Macro expansions</i>	5
0.1.4. <i>Repeat/While loop construct</i>	5
0.1.5. <i>Save and Restore operators</i>	5
0.1.6. <i>String expressions</i>	6
0.1.7. <i>Syntactic sugar</i>	6
0.1.8. <i>Command line switches</i>	6
0.1.9. <i>Error reporting</i>	6
0.1.10. <i>Bug Fixed in release 2.1 build 231</i>	6
0.2. DOCUMENT CONVENTIONS.....	7
0.3. ADDITIONAL RESOURCES	7
1. FOREWORD: WHAT IS UNIMAL?.....	8
2. QUICK START WITH UNIMAL	9
2.1. TABULATING A FUNCTION: UNIMAL LOOPS AND BUILT-IN MATH	9
2.2. PARAMETERS SHARING AMONG LANGUAGES: EXPORT STATEMENT	10
2.3. SOFTWARE DISTRIBUTION: IF STATEMENT	11
2.4. FURTHER APPLICATIONS	12
2.5. HIGHLIGHTS OF OTHER UNIMAL LANGUAGE FEATURES.....	12
2.5.1. <i>Composite names</i>	12
2.5.2. <i>Macros</i>	13
2.6. PUBLISHING THE INDICES OF ARRAY ENTRIES TO A HEADER FILE.....	14
2.6.1. <i>A useful design pattern</i>	14
2.6.2. <i>Export Push/Pop and string expressions</i>	15
2.6.3. <i>Include statement</i>	17
2.7. TOWARD TRULY REUSABLE MACROS: INSPECTING PROPERTIES OF THE ARGUMENTS	17
3. INVOKING UNIMAL	20
3.1. -I AND -I OPTIONS	20
3.2. -O AND -O OPTIONS.....	21
3.3. -D, -D OPTIONS	21
3.4. -P OPTION	22
3.5. -N OPTION.....	22
3.6. -S OPTION	22
3.7. -F OPTION	22
3.8. DEFAULT OUTPUT	23
3.9. ERROR REPORTING: UNIMAL.ERR.....	23
4. UNIMAL LANGUAGE REFERENCE GUIDE	24
4.1. GENERAL.....	24
4.2. LITERALS.....	24
4.2.1. <i>Numbers</i>	24
4.2.2. <i>Strings</i>	25
4.3. MACRO PARAMETERS (COMPILE-TIME VARIABLES)	25
4.4. SIMPLE NAMES.....	25

4.5.	FORMATS	26
4.5.1.	%s.....	26
4.5.2.	%d, %u, %x, %X	26
4.5.3.	%n	26
4.6.	COMPOSITE NAMES	27
4.7.	TARGET LANGUAGE INTERFACE.....	28
4.8.	NUMERIC EXPRESSIONS	29
4.8.1.	<i>Terms and operations</i>	29
4.8.2.	<i>Arithmetic expressions</i>	29
4.8.3.	<i>Shift expressions</i>	30
4.8.4.	<i>Bitwise logic expressions</i>	30
4.8.5.	<i>Logical expressions</i>	30
4.8.5.1.	Comparisons.....	31
4.8.5.2.	Negation '!'.....	31
4.8.5.3.	Logical AND '&&' and OR ' ' expressions.....	31
4.9.	STRING EXPRESSIONS	32
4.10.	ATTRIBUTES	32
4.11.	BUILT-IN EXPRESSIONS	32
4.11.1.	<i>Logical built-ins</i>	32
4.11.1.1.	Defined.....	32
4.11.1.2.	Isconst	32
4.11.2.	<i>Numeric functions</i>	33
4.11.2.1.	Math functions.....	33
4.11.2.2.	Misc. functions.....	34
4.11.3.	<i>String expressions</i>	34
4.11.3.1.	Name-to-string conversion	34
4.11.3.2.	Substring extraction <code>uSubstr</code>	34
4.11.3.3.	Concatenation <code>uJoin</code>	35
4.11.3.4.	Simplified Concatenation.....	35
4.11.3.5.	Splitting a string <code>uSplit</code>	36
4.11.3.6.	Defined encoding	36
4.12.	UNIMAL OPERATORS	37
4.12.1.	<i>Empty operator</i>	37
4.12.2.	<i>For</i>	38
4.12.3.	<i>Endfor</i>	38
4.12.4.	<i>Repeat</i>	38
4.12.5.	<i>While</i>	39
4.12.6.	<i>If</i>	39
4.12.7.	<i>Else</i>	39
4.12.8.	<i>Endif</i>	39
4.12.9.	<i>Ifdef</i>	39
4.12.10.	<i>Set</i>	40
4.12.11.	<i>Setstr</i>	40
4.12.12.	<i>Macro</i>	40
4.12.13.	<i>Endm</i>	41
4.12.14.	<i>Expand (non-recursive)</i>	41
4.12.15.	<i>Expand (possibly recursive)</i>	42
4.12.16.	<i>Include</i>	42
4.12.17.	<i>Export</i>	43
4.12.18.	<i>End</i>	43
4.12.19.	<i>Undef</i>	44
4.12.20.	<i>Save</i>	44
4.12.21.	<i>Restore</i>	44
4.13.	A USEFUL SHORTHAND FOR A LIST OF ARGUMENTS.....	45

4.14.	SPECIAL MACRO PARAMETERS	46
4.14.1.	<i>uAutoLine</i>	46
4.14.2.	<i>uAutoLineOut</i>	47
5.	ERROR DETECTION AND RECOVERY	48
5.1.	ERROR LOGGING MECHANISM IN UNIMAL.....	48
5.2.	UNIMAL ERROR MESSAGES REFERENCE.....	48
5.2.1.	<i>Default format of an error message</i>	48
5.2.2.	<i>Error message format mimicry</i>	49
5.2.3.	<i>F type: Fatal file errors</i>	50
5.2.3.1.	Error 0102.....	50
5.2.3.2.	Error 0103.....	50
5.2.3.3.	Errors 0104, 0105 (output file), 0106 (input file)	50
5.2.3.4.	Error 0111.....	50
5.2.4.	<i>Special F type error (Usage syntax)</i>	50
5.2.5.	<i>A type: Out of memory</i>	51
5.2.6.	<i>S type: Syntax errors</i>	51
5.2.6.1.	Error 2000 (The file has an unbalanced beginning or end of a block).....	51
5.2.6.2.	Error 2001 (General syntax error)	51
5.2.6.3.	Error 2002 (Macro redefinition)	52
5.2.6.4.	Error 2004 (Missing actual argument)	53
5.2.6.5.	Errors 2005, 2006, 2007 (Unmatched block operators)	53
5.2.6.6.	Error 2009 (Bad macro reference)	53
5.2.6.7.	Error 2010 (Unexpected type).....	53
5.2.6.8.	Error 2011 (undefined parameter)	54
5.2.6.9.	Error 2012 (formatting in composite names or target language interface)	54
5.2.6.10.	Error 2013 (expected macro parameter)	54
5.2.6.11.	Error 2014 (expected a numeric)	54
5.2.6.12.	Error 2015 (undefined string expression)	54
5.2.6.13.	Error 2016 (invalid string expression)	54
5.2.6.14.	Error 2017 (wrong number of arguments to a function)	55
5.2.6.15.	Error 2018 (literal number too large)	55
5.2.6.16.	Error 2019 (string not terminated)	55
5.2.6.17.	Error 2020 (Nested macro definition).....	55
5.2.6.18.	Error 2021 (Unmatched While)	55
5.2.6.19.	Error 2022 (Recursive macro expansion).....	55
5.2.7.	<i>L type: Lexical errors</i>	56
5.2.8.	<i>M type: Math errors</i>	56
5.2.8.1.	Errors 3500, 3501, 3502 (Arithmetic overflows).....	56
5.2.8.2.	Error 3503 (Divide by zero)	56
5.2.8.3.	Error 3504 (Non-positive divisor in remainder operation).....	56
5.2.8.4.	Errors 3510, 3511, 3512, 3513 (math functions errors)	56
5.2.9.	<i>Internal errors</i>	57
6.	ADDITIONAL FACTS	58
6.1.	(No) IMPLEMENTATION LIMITS	58
6.2.	TOKENIZATION	58
6.3.	INCLUDING AN OUTPUT FILE	58

0. Preliminary Notes

0.1. What's New in Unimal 2.1

(You can skip this change log if you are new to Unimal.)

0.1.1. Motivation

Generally, Unimal is reluctant to add new features because it wants to be very easy to learn. However, there is a balance to be found between being easy to learn and being easy to apply.

The features added to 2.1 come from the real-world experience and improve Unimal's ease of use.

0.1.2. Macro definitions

Beginning with version 2.0b, Unimal addresses two usability issues related to the placement of macro definitions:

- If *the same* macro definition is encountered more than once (as is the case with including the same file more than once), it is not an error.
- A macro definition can be placed within a block (i.e., between `If/Endif`, or `Else/Endif`, or `For/Endfor`); blocks are counted as if any macro expansions were completed. The primary motivation for this feature is simply to enable guarded include files (which technique is common practice in C and C++), but there are other uses of it, too.

Correspondingly, the error message S2008 is removed and a new error, [S2020](#) "Nested macro definition" is added.

0.1.3. Macro expansions

In version 2.1, a traditional macro invocation generates an error if it would cause a recursion. (Prior to 2.1, any such recursion would implicitly be infinite because Unimal must expand false blocks e.g. in a search for unbalanced `Endfor`.)

To allow recursive macro expansion, the user must indicate that the macro (including macros contained in it) is balanced; the syntax to do so is to pass the argument list in square brackets.

0.1.4. Repeat/While loop construct

Beginning with Unimal 2.0b, a `Repeat/While` loop construct analogous to C `do/while` construct is available. It doesn't add any new functionality because it can be emulated with a `For/Endfor` loop and manual manipulation of the loop counter. However, Unimal code in many cases becomes so much cleaner with the `Repeat/While` loop that its addition felt justified.

0.1.5. Save and Restore operators

Beginning with version 2.1:

Unimal now allows to stash a macro parameter away and to restore it, selectively, if so desired. E.g.

```
#MP Save myparam
```

The `Restore` operator restores the value(s) saved.

0.1.6. String expressions

Beginning with version 2.1:

- For a numeric expression `n`, `[n]` is a single-character string with its character encoded with the value of the least significant byte of `n`. E.g. if `n=65` and ASCII encoding is in effect, `[n]` is `"A"`.

0.1.7. Syntactic sugar

Beginning with version 2.1:

- In `Set`, `Compute` and `Setstr` operators, the equal sign may be omitted
- In `Setstr` operators, string concatenation can take a form of a sequence of string expressions, optionally separated by the `+` sign.

Beginning with version 2.0b:

- In a macro invocation, the keyword `Expand` may be omitted, and if the argument list is empty, the (empty) parentheses may be omitted, too.

0.1.8. Command line switches

Beginning with Unimal 2.1:

`-f<file>` reads the command line arguments from `<file>`; this serves as a command line extension

`-N<name>=<number>` added in version 2.0c is now checked for correctness

A new error message, F0111, is added to indicate an incorrect command line option

Beginning with Unimal 2.0c:

`-S<name>=<string>` defines macro parameter `<name>` with the string value `<string>`

`-N<name>=<number>` defines macro parameter `<name>` with the numeric value `<number>`

`-v` Displays version information

0.1.9. Error reporting

Prior to release 2.1 (build 231), Unimal did not log more than one error per line of input. That's because a second error is likely to be induced by the first one.

Beginning with release 2.1 (build 231), Unimal outputs all errors. Even though some errors are induced by a previous error, the error output allows better understanding of the root cause of the failed execution.

0.1.10. Bug Fixed in release 2.1 build 231

- This bug is unique to Unimal 2.1 build 227.
 - `-v` and `-p` command-line options are occasionally not recognized
- A dangling end-of block by the end of a file may cause Unimal to crash. Now it aborts processing cleanly, with a sensible error message
- Errors were suppressed if they were found in a false block. That was wrong e.g. for file access errors. Now applicable errors are reported always.

0.2. Document conventions

Code snippets, names of variables and such are in `courier new` font.

Of them, keywords are `blue`.

Formal arguments of Unimal macros are shown `#red#`.

Unimal comments are shown in `green`.

Literal strings are `shown in brown`.

When line numbers in a code snippet are referenced, the line numbers in parentheses are shown to the left of the code lines.

0.3. Additional Resources

There is a `QuickStart` subdirectory of the `Samples` directory in the distribution; it contains the files referenced in the Quick Start section. If you prefer to work along, do use those files for reference.

Also, don't miss the Application Notes; some are included in the distribution; more are available on the Web site.

1. Foreword: What is Unimal?

In short, Unimal is an advanced language-independent macro preprocessor. That is, it is a utility that processes a source file into one (or more!) output file(s) which are, in their turn, source file(s) in one (or more!) programming languages.

The name stands for UNIfied MAcro Language. 'Unified' here means that the same macro processor applies to various (programming) languages, like C, Assembler, Linker command language, make files, or almost any language whatsoever.

Unimal is not made to replace any software development tools; it is to supplement them with better software management capabilities.

A software developer would use Unimal in a situation where there is a need in a powerful macro processor. The powerful features of Unimal, some unique, make it a macro processor of choice.

Unimal is not (maybe, unfortunately) designed with utmost elegance in mind. Instead, it is designed to be simple in accomplishing simple tasks and powerful enough to make complex solutions possible.

The next section introduces several practical problems and illustrates Unimal features facilitating the solutions.

2. Quick Start with Unimal

Conceptually, Unimal macro language is very simple: it is line-based, and each line in a source file is either a line in a target programming language or a Unimal statement. The former can have a special markup, which instructs Unimal to replace the markup with corresponding text.

2.1. Tabulating a function: Unimal loops and built-in math

As an illustration, consider the problem of tabulating a hard-to-compute function in a constant integer array – a problem often encountered in embedded programming. In our first example, we want to tabulate, in C, a scaled sine wave, $10000 \cdot \sin(x)$ at seven equidistant points in the segment $[0, \pi/2]$. Here is a solution:

```
(1) const int sinewave[] = {  
(2) #MP For n=0, 6  
(3) #MP      val = Usin(10000, 1, n, 2*6)  
(4) #mp%dval,  
(5) #MP Endfor  
(6) };
```

The Unimal output is

```
const int sinewave[] = {  
    0,  
    2588,  
    5000,  
    7071,  
    8660,  
    9659,  
    10000,  
};
```

(If you prefer to work along, run
Unimal sine6.u

from the command line in the QuickStart folder; the result will be sent to the standard output.)

Here's what is happening here:

Line 1 is normal C code; since it doesn't have any Unimal markup, it is copied from input to output verbatim.

Line 2 starts with the marker `#MP`; this indicates a Unimal statement. As with *all* Unimal statements, nothing from this line is sent to output. The statement itself is a loop statement; it instructs Unimal to re-scan the source file until the matching end of loop for `n=0, 1, ..., 6`. Matching end of loop is found in line 5, so lines 3 and 4 will be scanned for all values of `n`.

Line 3 (again, because of the `#MP` marker) is a Unimal statement. It is a Set statement, assigning the value of a numeric expression to the *macro parameter* `val`. In our case, the numeric expression is a built-in function, `Usin`, which computes a scaled integer sine:

$$\text{Usin}(a, b, c, d) = \frac{a}{b} \cdot \sin\left(\pi \frac{c}{d}\right)$$

Line 4 does not start with the #MP marker, so it is considered a line in the target language and is to be sent to the output. However, it contains a Unimal markup, #mp%d, which instructs Unimal to render the macro parameter that follows (val) as a decimal number. What's outside the markup (in this case, just a comma) is copied to the output unchanged. As n changes with each rescan of the source, so does val. This is how the numbers in the output are produced.

Line 5 is a Unimal statement (it starts with #MP); this is the end-of-loop statement which indicates the boundary for repeated source rescan.

Finally, line 6 is a target language statement without markup; it is copied to the output.

This simple example hints at Unimal's powers in automating static (compile-time) initialization. While this is most important in resource-constrained embedded applications, "normal" computer applications can benefit from Unimal in implementing table-driven algorithms and/or software designs.

2.2. Parameters sharing among languages: Export statement

As a different example, consider sharing constant parameters across languages. Let's say we need to share a symbolic definition, which, say in an Assembler looks like

```
MYDATA .equ 17
```

and in C,

```
#define MYDATA 17
```

For the purpose of project maintenance, we want to enter one definition once (or else they are going to diverge). How to do this if C doesn't understand Assembler definitions and vice versa?

Let's have Unimal make an Assembler include file, mydata.inc, and a C header file, mydata.h, from a single Unimal definition:

Example: Sharing a definition between C and Assembler

```
#MP Set MYTHING = 17 ;MYTHING is a Unimal name
```

```
#MP Export (0) "mything.h"
```

```
#define MYTHING #mp%dMYTHING
```

```
#MP Export (0) "mything.inc"
```

```
MYTHING .equ #mp%dMYTHING
```

```
#MP Export (0) ""
```

Done exporting MYTHING to mything.inc and mything.h

(If you are working along, c-asm.u is the file.)

The first line, since it doesn't begin with #MP, is considered in a target language; in this case it is English. It is sent to the output which, by default, is standard output stream (stdout).

The second line is an already familiar Set operator. Note the keyword Set (which is optional, as we have seen). As a result, Unimal *macro parameter* MYTHING is assigned the numeric

value 17. This is a common, language-independent definition which we want to be entered only once. Any text from a semicolon to the end of line is a comment and is ignored.

The third line is the Unimal export statement: the argument in parentheses, being a zero, instructs Unimal to overwrite the output file if it exists. The name of the output file is supplied as the second argument, so from the next line on the output is sent to the file `mything.h`.

Line 4 is a target language interface with a markup telling Unimal to render the macro parameter `MYTHING` as a decimal number. The resulting line,
`#define MYTHING 17`
 is sent to `mything.h`

Similarly, line 5 switches output to the file `mything.inc`, and line 6, transformed to
`MYTHING .equ 17`
 is sent to `mything.inc`

Line 7 switches the output again, but this time the file name is an empty string. By convention, it means the default output (`stdout`), and line 8 is output there.

A simple yet important application of parameter sharing in the embedded world is sharing the microcontroller memory map among the programming language(s), linker command file, and other post-link tools, like program image CRC calculations.

2.3. Software distribution: *If statement*

Unimal can be very useful in configuring and managing multiple projects in a family. As an almost trivial example, consider a family of projects with customer-specific features implemented as fragments of code. Using C/C++ preprocessor, you might write something like this:

```
#define CUSTOMER CUSTOMER_B
.....
#if CUSTOMER==CUSTOMER_A
<customer A code>
#endif
#if CUSTOMER==CUSTOMER_B
<customer B code>
#endif
#if CUSTOMER==CUSTOMER_C
<customer C code>
#endif
```

If, however, you deliver your product in source code, you don't necessarily want a customer to see what other customers are getting (or who they are or that they even exist). Unimal allows writing a similar thing:

```
#MP Set CUSTOMER = CUSTOMER_B
.....
#MP If CUSTOMER==CUSTOMER_A
```

```
<customer A code>
#MP Endif
#MP If CUSTOMER==CUSTOMER_B
<customer B code>
#MP Endif
#MP If CUSTOMER==CUSTOMER_C
<customer C code>
#MP Endif
```

This snippet illustrates the If statement; it works as intuitively expected. The result of Unimal processing of this fragment will contain only the code for the selected customer, in this example,

```
<customer B code>
```

2.4. Further applications

Unimal can be extremely useful in numerous situations a programming practitioner faces. A few simpler ones are discussed above; others may be mentioned further as we go along.

Cases that are more realistic may be more complex than this introduction is comfortable with; they are covered in separate application notes.

One of the more interesting (and algorithmically complex) applications involves automatic generation of auxiliary data, such as lookup tables, along with the corresponding accessor functions.

However, technically simple applications, like forced loop unrolling, or compacting data representation, are no less useful.

And that's the whole point of Unimal:

You decide what is to be achieved at build time. You, not bound by limits of your programming language, come up with a conceptual algorithm of realizing your goal. Unimal provides expressive power to implement your algorithm.

2.5. Highlights of other Unimal language features

2.5.1. Composite names

An interesting feature of Unimal is support of *composite names*. They provide functionality of arrays, sparse arrays, and associative arrays (like Perl hashes) in a uniform manner. As a simple example, consider a disparate set of, say, character strings, which you would like to process later in a loop, so you would like to give them "indexed" names.

The following code does just that:

```
(1) #MP count = 0
(2) #MP Setstr str%dcount = "foo"
(3) #MP count = count+1
(4) #MP Setstr str%dcount = "bar"
(5) #MP count = count+1
(6) #MP Setstr str%dcount = "baz"
```

```
(7) #MP count = count+1
```

In lines 2, 4 and 6 we see a common construct, `str%dcount` (and a `Setstr` operator assigning a string value to a macro parameter). It is a base name (in our case, `str`), followed by one or more *suffixes*. A suffix is a format (such as `%d`) followed by a simple name (`count`). A base name is a simple name or it can be empty.

When we arrive at line 2, `count=0`, and the name in `Setstr` is rendered by “printing” the suffix, according to the format, as a decimal number, resulting in `str0`. When we are at line 4, `count` is a 1, so line 4 is equivalent to

```
#MP Setstr str1 = "bar"
```

Similarly, line 6 is equivalent to

```
#MP Setstr str2 = "baz"
```

As a useful side effect, after line 7, `count` is the number of strings we enumerated.

2.5.2. Macros

Unimal allows shaping this in a prettier and better maintainable way by using its macro facility.

In general, our definitions will have some initialization (like line 1 above), actual definition statements and perhaps some post-processing stuff as well. So, a typical data definition would look like

```
#MP Expand BeginData ()
#MP Expand DefineData ("foo")
#MP Expand DefineData ("bar")
#MP Expand DefineData ("baz")
#MP Expand EndData ()
```

The Unimal operator `Expand` expands a named macro with *arguments* passed in a comma-separated list in parentheses, which may be empty. A macro name can be the name of a previously defined macro or a *string expression* which resolves to a name of a previously defined macro.

The keyword `Expand` is optional and may be omitted without restrictions. Empty parentheses may be omitted, too. So, the text above may be rewritten as

```
#MP BeginData
#MP DefineData ("foo")
#MP DefineData ("bar")
#MP DefineData ("baz")
#MP EndData
```

It is purely the matter of style to choose one look over the other.

Let's define the macros applicable to this example:

```
(1) #MP Macro BeginData ; ()
(2) #MP count = 0
(3) #MP Endm
(4) #MP Macro DefineData ; (string)
(5) #MP Setstr str%dcount = #1#
(6) #MP count = count+1
```

```
(7) #MP Endm
(8) #MP Macro EndData ;()
(9) #MP NumStrings = count
(10) #MP Undef count {NUM}
(11) #MP Endm
```

A macro definition begins the keyword `Macro`, followed by a simple name which is the name under which the macro will be known. It ends with the keyword `Endm`. Any lines in between are the *macro body* which is substituted for the `Expand` operator. In the Unimal text above, lines 1, 4, 8 are beginnings of macro definitions, lines 3, 7, 11 are their respective ends, and lines 2, 5-6, 9-10 are the corresponding bodies.

Note that the number and types of arguments are *not* part of a macro definition; so if you write a macro expecting certain parameters, it is a good idea to indicate that in a comment.

In the second macro body, we encounter the expression `#1#`; it is a way to name the first argument passed to the macro.

In the third macro body, just for the sake of illustration, we save the number of strings defined and *undefine* a numeric (`NUM`) value of `count`. If `count` had a string and/or a macro value, they would still be defined. That is to say, a Unimal macro parameter can have a numeric, a string and a macro values at the same time (one might say they belong to different namespaces; which value is used depends on the context).

We can test our macros by printing the enumerated strings (see `enums.u`):

```
We enumerated #mp%dNumStrings strings:
#MP For i=0, NumStrings -1
#mp%i. "#mp%sstr%i"
#MP Endfor
```

Here is the output:

```
We enumerated 3 strings:
0. "foo"
1. "bar"
2. "baz"
```

2.6. Publishing the indices of array entries to a header file

2.6.1. A useful design pattern

Typically, a definition sequence `Begin/Define/End` is meant to process the `Define` type statements more than once, i.e. in a loop. This requires wrapping the `For` statement in the `Begin` macro and the `Endfor` statement in `End`. Unimal allows this spanning of a loop (as well as of `If/Else/Endif` compounds) across several macros.

As a simple example, consider defining an initialized array `foo` of some objects in a C file `foo.c`, and publishing their indices in a header `foo.h`.

This is what it might look like:

```
#MP Expand BeginArray("foo")
#MP Expand ArrayEntry("MYINDEX", "myObject")
#MP Expand ArrayEntry("YOURINDEX", "yourObject")
#MP Expand ArrayEntry("HISINDEX", "hisObject")
#MP Expand ArrayEntry("HERINDEX", "herObject")
#MP Expand EndArray()
```

BeginArray takes the base name of a file (before extension); it is also the name of the array to generate. ArrayEntry takes a symbolic name of the index and an object expression.

2.6.2. Export Push/Pop and string expressions

Here are the macro definitions with explanations that follow (see `indices.u`):

```
#MP Macro BeginArray ;(basename)
#MP Setstr suffix0 = ".c"
#MP Setstr suffix1 = ".h"
#MP Export Push
#MP For pass = 0,1
#MP     Export (0) {uJoin, #1#, suffix%dpass}
#MP     count = 0
#MP If pass == 0
ob_type #mp%s#1#[] = {
#MP Endif
#MP Endm

#MP Macro ArrayEntry ;(index_name, object_name)
#MP If pass == 0
    #mp%s#2#,
#MP Endif
#MP If pass == 1
#define #mp%s#1# #mp%dcnt
#MP Endif
#MP count = count + 1
#MP Endm

#MP Macro EndArray ;()
#MP     If pass == 0
};
#MP     Endif
#MP     Undef suffix%dpass {STR}
#MP Endfor
#MP Undef count {NUM}
#MP Undef pass {NUM}
#MP Export Pop
```

#MP Endm

BeginEntry first defines two strings, suffix0 and suffix1, to be the extensions of the files we are going to make. Second, the Export Push statement remembers the current output file to restore after we are done; it is a clean way to go since we are going to change the output file. The next (For) statement starts the loop which will be repeated twice, for pass values 0 and 1. The loop will span quite a few statements until Endfor is encountered (in EndArray).

The next statement (which is already within the loop body) changes the output. The (0) argument means that if the file exists, it will be overwritten. The second argument is the filename; it is represented here by a string expression, {uJoin, #1#, suffix%dpass}.

The first in the list here is the name of string operation, in this case, concatenation. The second in the list is the first actual argument of the macro (like the "foo" that was passed in the example. The third in the list is a macro parameter suffix%dpass with a composite name. In it, the rendering format causes pass appear as 0 and 1 respectively in the two passes. So, in pass 0, suffix%dpass resolves to suffix0 (which we already cleverly set to ".c", and in pass 1 to suffix1 (".h"). So, the whole string expression evaluates to "foo.c" in pass 0 and to "foo.h" in pass 1.

The next line initializes count to 0; it will be counting the array entries identically in both loop passes. Finally, if pass=0 (we are making the .c file), we render the beginning of the array definition, like

```
ob_type foo[] = {
```

The macro ArrayEntry is designed to be scanned twice (in a loop) but is quite simple:

In pass 0 it renders the object expression taken verbatim from the second macro argument, followed by a comma to follow C initialization syntax, like

```
myObject,
```

In pass 1 it generates C symbolic definition of the index of the array entry, taking the symbolic name from the first macro argument, and the value, of course, is count, the ordinal number of the instance of ArrayEntry being processed, e.g.

```
#define MYINDEX 0
```

And, in both passes, count is incremented (to do the counting).

The EndArray, in pass 0, closes the C array according to the language syntax:

```
};
```

The next is a cleanup statement: in pass 0 we no longer need suffix0, and in pass 1, suffix1, so we *undefine* them so that they not be used elsewhere inadvertently. Note the attribute STR in braces: just in case, we undefine only the string value; if, say, suffix0 had a macro or a numeric value, they'd still remain defined.

The next statement (Endfor) ends the loop body that began far away in BeginArray.

As a final cleanup, EndArray undefines numeric values of count and pass and, in the Export Pop statement, restores the output to whatever stream it was before BeginArray.

If we run the test file (`Unimal indices.u`), the following files will be created:

`foo.c`

```
ob_type foo[] = {
    myObject,
    yourObject,
    hisObject,
    herObject,
};
```

`foo.h`

```
#define MYINDEX 0
#define YOURINDEX 1
#define HISINDEX 2
#define HERINDEX 3
```

2.6.3. Include statement

It is worth noting that the macros we devised are entirely reusable. We can keep them in a separate *include file*, like `indexgen.u`. Then our toy application would begin with the statement

```
#MP Include "indexgen.u"
```

The example file is `indices1.u` (of course, together with `indexgen.u`). The outputs they produce are the same as above.

2.7. Toward truly reusable macros: inspecting properties of the arguments

As a final example, also illustrating some more Unimal built-ins, consider a macro which prints information about its arguments:

```
(1) #MP Macro Args ;(list)
(2) #MP      For i=0, #0#
(3) #MP      If Isconst(#i#)
(4) Argument ##mp%di is constant
(5) #MP      If Isconst(#i#{NUM})
(6)      Numeric value: #mp%d#i#
(7) #MP      Endif
(8) #MP      If Isconst(#i#{STR})
(9)      String value: #mp%s#i#
(10) #MP      Endif
(11) #MP      Else
(12) Argument ##mp%di is NOT constant; its name is #mp%n#i#
(13) #MP      If Defined ( #i# {NUM} )
```

```

(14)      Numeric value: #mp%d#i# (0x#mp%08X#i#)
(15) #MP      Endif
(16) #MP      If Defined(#i#{STR})
(17)      String value: #mp%s#i#
(18) #MP      Endif
(19) #MP      If Defined(#i#{MAC})
(20)      It is a defined macro
(21) #MP      Endif
(22) #MP      Endif
(23) #MP      Endfor
(24) #MP Endm

```

The macro is a large loop starting at line 2; the upper limit, `#0#` is a special (and always defined) macro argument whose value is the number of actual arguments in the invocation of the macro.

Line 3 tests a built-in expression, `Isconst()`; its value is 1 if the argument *cannot* be changed within the macro, i.e. it is an expression, or 0 if the argument is a macro parameter (a compile-time variable). The argument to `Isconst` in line 2 is `#i#`; it is the argument number `i` in the macro invocation.

If the test in line 3 passes, line 4 prints a message to that effect, referencing the ordinal number of the argument.

Then, lines 5 and 8 do more detailed tests: whether the argument is a number or a string; lines 6 and 9 print the values. Note how the arguments to `Isconst` are qualified with *attributes*. (Exactly one of the two tests must pass.)

If the test in line 3 fails, we are into the `Else` clause in line 11.

Line 12 prints a corresponding message indicating the ordinal number of the argument (using the `%d` format) and the name of the argument (using the `%n` format).

Then, lines 13, 16, 19 test whether the argument (which is already known to be a parameter, or variable) has a numeric, string, and/or macro value. This is done using a built-in `Defined()` expression, which is a 1 if the corresponding qualified name is defined (i.e. has a value corresponding to the attribute), and is a 0 otherwise. An unqualified `Defined`, like `Defined(x)` is shorthand for `Defined(x{NUM,STR})`.

If a test in line 13, 16, or 19 passes, a corresponding message is printed by lines 14, 17, 20. Note that since a named parameter can have values with different attributes, more than one test can pass.

To test the macro, let's assign a string and a numeric value to the same name as that of the macro:

```

#MP Setstr Args = "abcd"
#MP Args = 2006

```

Now, let's invoke it:

```

#MP Expand Args(Args, (Args), {Args}, !Defined(foo), {uSubstr, Args, Ustrlen(Args), 0})

```

The second argument is a numeric expression (same value as the numeric value of `Args`). The third argument is a string expression (same value as the string value of `Args`).

The fourth argument is a logical negation of an unqualified Defined expression, and it is a numeric expression like in C.

The fifth argument is a string expression. `uSubstr` extracts from its string second argument a substring delimited by the indices represented by the numeric third and fourth arguments. If start index is greater than the end index, the order of characters is reverted.

Here is the output of the test invocation:

```
Argument #0 is constant
    Numeric value: 5
Argument #1 is NOT constant; its name is Args
    Numeric value: 2006 (0x000007D6)
    String value: abcd
    It is a defined macro
Argument #2 is constant
    Numeric value: 2006
Argument #3 is constant
    String value: Args
Argument #4 is constant
    Numeric value: 1
Argument #5 is constant
    String value: dcba
```

3. Invoking Unimal

Unimal is a command-line utility. Its invocation syntax is

```
Unimal [<options>] <input_file> [<options>]
```

(As usual, brackets indicate optional parameters.)

<input_file> is a Unimal source file.

The following options are supported:

- `-i<Path>`, `-I<Path>` specify search directories for include files
- `-o<Path>`, `-O<Path>` specify output directories
- `-D<Filename>`, `-d<Filename>` specify the file where include files dependencies are dumped
- `-p` changes the format of filenames maintained by Unimal
- `-v` prints version information and options help to stderr (standard error device, typically the console). Additionally, each output file name is printed to stderr when that output becomes current.
- `-N<name>=<value>` defines a named macro parameter with the specified numeric value
- `-S<name>=<string>` defines a named macro parameter with the specified string value
- `-f<filename>` reads the command line options from the named files

Note that no space is allowed between an option and a path- or file- or parameter name.

Unimal invoked incorrectly terminates immediately with a fatal error (F0099 or F0111).

3.1. `-i` and `-I` options

These options apply to Unimal Include statements where the filename to include either has no path part or is a relative pathname.

In this case, the directories specified by an `-i` or `-I` options are searched in the order of their appearance on the command line until the matching include file is found.

The difference between `-i` and `-I` is where the specified directory has itself a relative path. In this case, `-IPath` treats Path as relative to the current working directory at the time of invoking Unimal. On the contrary, `-iPath` treats Path as relative to the directory where the file currently being processed is located. For instance, consider a command line

```
Unimal foo.u -ibar -ibaz
```

Let's assume that `foo.u` has a statement

```
#MP Include "foo.uu"
```

and `foo.uu` is found in `baz` directory. If `foo.uu`, in turn, includes `fuu.uu`, the latter is searched in directories relative to `baz`, i.e., in `baz/bar` and `baz/baz`.

Typically, you would use only one of the `-i` and `-I` options, depending on how your project directories are designed. The two options are provided for flexibility and ease of integrating Unimal in your build process.

Any number of `-i` and/or `-I` options can be supplied on the command line. As usual, pathnames containing spaces must be quoted.

Notes:

1. The directory of the currently processed file is always searched first. It can be said that the `-i .` option (search current file-relative directory) is built in.
2. It matters whether `-i` or `-I` option is placed before or after the input file on the command line. Directories, which are specified *before* the input file, are searched for the input file itself, provided it is specified with the relative pathname. In *that* search, `-i` option, too, treats relative paths as relative to the current working directory. (Those familiar with VPATH functionality of GNU make, will notice similarity. For simplicity of integration in the project build process, Unimal has this functionality built in.)
3. On Windows platforms, you can specify a path relative to the CWD on a particular drive, such as `E:\foo`. Unimal does not treat these path specifications as relative, even if the specified drive is the current drive.

3.2. `-o` and `-O` options

These options specify the output directory for the cases where the output file in an `Export` statement has a relative pathname.

You *can* specify more than one `-o` and/or `-O` option, but only one (the last one) takes effect.

The difference between `-o` and `-O` is where the specified directory has itself a relative path. In this case, `-OPath` treats `Path` as relative to the current working directory at the time of invoking Unimal. On the contrary, `-oPath` treats `Path` as relative to the directory where the file currently being processed is located. If no `-o`, `-O` options are specified, then `-o.` assumed, and an output file will be located relative to the currently processed file. This might not be the best expected behavior, and the option `-O.` is often preferred in “descend into subdirectories” build processes.

Note:

1. Unimal doesn’t create directories; they must exist or Unimal will fail.

3.3. `-d`, `-D` options

For a change, `-d` and `-D` options are (almost) equivalent. They specify the dependencies file. If the filename is relative, it is considered relative to the current working directory at the time of Unimal invocation.

Since Unimal can include files, directly or indirectly, while processing the input file, it is subject to the curse of include files dependency: The build process needs to know whether a Unimal source file needs to be processed when an include file changes. To assist with the task, Unimal can create a list of fully qualified names of files included while processing the input file. The list is saved in the file specified by the `-d` or `-D` option.

The only difference between `-d` and `-D` options is on Windows platforms: `-d` produces native fully qualified pathnames (like `C:\foo\Bar`) but `-D` drops the drive letter and converts backslashes to slashes (like `/foo/Bar`) to produce Unix-like names. This is provided as assistance to multiplatform development with isomorphic project directories structures.

Notes:

1. Unimal doesn't create directories; if the dependency filename contains path, the path must exist or Unimal will fail.
2. The `-d`, `-D` options are optional; if not specified, no dependency file will be created. However, you *can* specify more than one option; the last one takes effect.

3.4. `-p` option

By default, file names maintained by Unimal for error messages and "automatic" macro parameters are in exactly same form as supplied in the source file(s) and the command line.

If, however, `-p` option is used, Unimal maintains fully qualified pathnames instead.

3.5. `-N` option

This option has a format `-N<name>=<value>`, e.g.

`-Nmyvar=-2007`

The effect of it is that the named macro parameter (e.g. `myvar`) gets the specified numeric value (e.g. `-2007`). The value must be a decimal number within the 32-bit range (see [Set operator](#)).

Any number of `-N` options is allowed; if they assign values to the same parameter, the last assignment takes effect

3.6. `-S` option

This option has a format `-S<name>=<string>`, e.g.

`-Smyvar=year2007ishistory`

The effect of it is that the named macro parameter (e.g. `myvar`) gets the specified string value (e.g. `year2007ishistory`). It is equivalent to

`#MP Setstr myvar="year2007ishistory"` in the source file.

Note that the string value itself is not quoted. However, the string visible to Unimal is what comes through the command interpreter (shell).

On Linux platforms it is safe to always quote the string and escape literal quotes in it with a backslash. E.g., from

`-Sfoo="bar: \"baz\""` on the command line Unimal will see

`-Sfoo=bar: "baz"` which is equivalent to

`#MP Setstr foo=#@ bar: "baz"#` in the source file.

Windows default shell, CMD.EXE, has some bizarre rules governing the quotes and it may refuse to strip them when you least expect it, e.g. when a string is invented by a makefile or your Integrated Development Environment. Be aware of this.

3.7. `-f` option

The `-f<filename>` option reads the command-line options from the named file. This eliminates any limitations on the length of the command line and the possible unpleasant dealings with the idiosyncrasies of the command interpreter.

There may be more than one `-f` option on the command line and the use of this option is allowed within an options file. However, recursive inclusion of options files is not allowed (and it is senseless anyway).

An options file is processed as follows:

- Each line in the file must be shorter than 5 K

- Each line is stripped of the leading white spaces
- If a stripped line is empty or starts with a semicolon (;) or a pound sign (#), it is considered a comment and is ignored
- A (stripped) non-comment line is considered to contain a single option
- Strings (parameter names, file names and directory names) are not quoted even if they contain spaces. Leading and trailing spaces, if any, are considered part of a string. Otherwise, trailing spaces are illegal. The first equal sign (=) separates the parameter name and the value for the options -S and -N. Trailing spaces are not allowed for the -N option.

Examples:

-N mine = 2 <EndOfLine> - invalid (has a trailing space)

-N mine = 2<EndOfLine> - a parameter oddly named " mine " (with a leading and a trailing space) gets the value 2

-IMy Best Folder Ever <EndOfLine> makes Unimal search the directory "My Best Folder Ever " (with a trailing space as written) for include files

My Source<EndOfLine> instructs Unimal to process the source file "My Source" (without leading spaces)

3.8. Default output

Unimal's output goes to the standard output device (STDOUT, typically, a console, unless the output is redirected) until Unimal encounters an Export statement. After that, the output is sent to a file or to a console as controlled by the source file.

3.9. Error reporting: *Unimal.err*

On completion, Unimal returns a return code, which is a zero if no errors were encountered. In case of errors, Unimal returns a non-zero value. For error details, please see the section on error detection and recovery.

Unimal produces a descriptive error message, typically, no more than one per line of the source to avoid over-reporting induced errors.

Identical error messages are output to one or more of the following:

- Standard error device (STDERR, typically, a console)
- The Unimal output (standard output or a file requested in an [Export](#) statement)
- The file `Unimal.err` which is created in the current working directory.

The file `Unimal.err` collects all errors in one place regardless of the number of output files.

4. Unimal language reference guide

4.1. General

Unimal is designed as a macro language extension of a target programming language, independent of that target language. Unimal pre-processor processes the input stream consisting of the input file and any include files encountered. It sends the output to the output stream, which is by default the standard output, but which can be redirected to one or more output files using the means of the Unimal language.

Unimal language consists of two inter-related parts: Unimal operators and Unimal target language interface.

Unimal target language interface consists of a special mark-up signatures highly unlikely to be met in a programming language, followed by a Unimal macro parameter (compile-time variable) equipped with instructions (format) how to send it to the output stream.

Unimal operators control the way the input stream is processed and calculate the macro parameters used both internally and in the target language interface.

Macro parameters are analogous to variables in programming languages, except that all values are known at processing (compile) time. All macro parameters have global scope within the file where they are defined, but they cannot be shared across the files other than by means of shared include files.

4.2. Literals

4.2.1. Numbers

Unimal supports decimal and hexadecimal number literals.

A decimal number is represented by digits 0 to 9 and must be in the range from 0 to 2,147,483,647. (Unlike C, a decimal number can start with a 0.)

Examples:

01234, 1717

A hexadecimal number is represented by digits 0 to 9 and A to F (case-insensitive) and is preceded by 0X (or 0x). It must be in the range 0x00000000 to 0xFFFFFFFF. The numbers in the range 0x00000000 to 0x7FFFFFFF are considered positive, and in the range 0x80000000 to 0xFFFFFFFF, negative in 2's complement representation.

Examples:

0xab, 0XCDEF

Notes:

1. Please note that e.g. -5 is *not* a literal: it is an expression. Expressions are described further below. The distinction matters very little though.
2. An odd consequence is that the number -2147483648, which is the smallest numeric value Unimal supports, cannot be used literally. A workaround is to use -2147483647-1 or 0x80000000.

4.2.2. Strings

String literals in Unimal are any sequences of characters enclosed between the left and the right delimiter. One pair of delimiters is very common: both left and right delimiters are double-quotation marks (`"`). A literal quotation mark character in a string is represented by doubling the character. E.g.:

- `"Unimal"` is a literal string representing the text *Unimal*
- `""Unimal"" is a macro processor"` is a literal string representing the text *"Unimal" is a macro processor*

There is a known problem with any method of escaping the delimiter character; anyone who wrote, say, inline sed command within a makefile knows it. The problem is that a string may be a part of a literal in a programming language or of a shell command; the literal delimiter there must be escaped, so in a second-order language it must be escaped twice, etc. This quickly makes the string absolutely unreadable.

Unimal addresses this problem by supporting a pair of quite unusual delimiters: `#@` and `#`. Any ``#'` character that is a part of the string must be doubled.

Examples:

```
#@ Unimal#           #@Problem set ## 11#
```

Up until now, we avoided a question of what a character is.

In Unimal, a character is whatever symbol you can type in your programmer's editor. (This will almost universally include the ASCII character set, unless you are working on an EBCDIC machine.) Note that if your editor supports a multi-byte character encoding, such as UTF8, then such characters are valid Unimal characters but may appear differently in a different editor.

4.3. Macro parameters (compile-time variables)

Unimal macro parameters (sometimes referred to as compile-time variables) are named objects which can hold:

- a numeric value in the range -2,147,483,648 to 2,147,483,647, and/or
- a character string of single-byte characters, and/or
- a macro definition

What are the legitimate names for macro parameters is described further below.

Note that a macro parameter can hold a value of different "types" at the same time; which value is used depends on the context.

See also:

[Macros](#), [simple names](#), [composite names](#)

4.4. Simple Names

Simple names in Unimal are words composed with letters A-Z, a-z, digits 0-9 and the underscore ``_'`, and cannot start with a digit. Names in Unimal are case sensitive. Examples:

```
_01234           AtoZ
```

Example with errors:

```
0string          1step (starts with a digit)
str!ng           (contains an illegal character)
```

A very special type of a simple name is a decimal number between 0 and 999 enclosed between the '#' characters. This type of names is allowed only inside a macro body and is used to denote a formal argument, which is replaced with the corresponding actual argument during macro expansion. Examples:

#003# - third formal argument,

#99# - ninety-ninth formal argument.

Note that leading zeros have no effect; it's the numeric value that matters.

#0# has a special meaning: it is the number of actual arguments passed to the macro.

See also:

[Macros](#), [composite names](#)

4.5. Formats

Formats are special character strings used for rendering Unimal macro parameters. They are used both in Unimal statements (operators) and in the target language interface. Unimal supports the simplest formats of C `printf` variety and an additional format.

4.5.1. %s

The format `%s` is used to render the string value of a macro parameter. If the actual parameter does not have a string value, Unimal generates an error. Example:

`%s#1#`

produces a string passed as the first macro argument.

See also:

[Macros](#), [Target language interface](#).

4.5.2. %d, %u, %x, %X

The `%d`, `%u`, `%x` and `%X` formats are numeric formats used to render numeric macro parameters in printable format. `%d` renders a (signed) decimal number, `%u`, an unsigned decimal number, and `%x` and `%X`, an unsigned hexadecimal number. The difference between `%x` and `%X` is that `%x` prints hex digits A – F in lower case, and `%X`, in upper case.

It's worth noting that Unimal prints the value rather blindly, without respect to the arithmetic value. For instance, a macro parameter with a value `-1` is printed as `-1` with `%d` and as `4294967295` with `%u`.

All four of these formats have variants with a `0<width>` between a '%' and a format letter, where `<width>` is a digit from 1 to 9. Such a variant will print a numeric value padded with zeros so that it has `<width>` digits. If the value is such that when printed has more than `<width>` digits, no padding occur. In this context, a '-' in a negative number counts as a digit. Example: a value `-29` is printed with `%04u` as `4294967267`, with `%04d` as `-029` and with `%04x` as `ffffffe3`.

See also:

[Composite names](#), [Target language interface](#), [Macros](#)

4.5.3. %n

The format `%n` is used to render the name of a macro parameter.

This is handy, among other things, in debugging macros. For instance, a [target language line](#) in a macro,

```
#mp%n#1# #mp%d#1#
```

will print the name and the numeric value of the macro's actual argument 1.

See also:

[Composite names](#), [Target language interface](#), [Macros](#)

4.6. Composite names

A composite name is an optional simple name followed with one or more macro parameters with simple names rendered with any of the formats. Those rendered macro parameters must be already defined. A more formal definition:

```
<composite_name> ::=
    <format><simple_name> |
    <simple_name><format><simple_name> |
    <composite_name><format><simple_name>
```

Example:

```
x%uX%06xX
```

is a valid composite name, and names different macro parameters, depending on the numeric value of x. If x is a 17 (11 hex) then x%uX%06xX references x1700000011.

Another example:

```
%sX
```

is a valid composite name, and names different macro parameters, depending on the string value of x. If x has a string value "Unimal is a macro processor!" then %sX references a macro parameter named

```
Unimal is a macro processor!
```

It is worth noting that such a name is not a valid simple name: it contains illegal characters (spaces and an exclamation point). So, it cannot be used as a literal (simple) name. Yet, it is a valid composite name, and it can be referenced by a different name composition.

For instance, if X1 and X2 have string values "imal is a macro " and "processor!" respectively, then the composite name Un%sX1%sX2 references the same macro parameter as %sX, namely,

```
Unimal is a macro processor!
```

The last example:

```
foo_%nbar
```

This simply references a name foo_bar. This in itself has very little value except in context of a macro definition, where construct foo_%n#1# references a parameter according to the name passed as the first argument in the macro invocation.

Notes:

1. If a composite name is passed as an actual argument to a macro, it is first resolved to a simple name and is treated as such inside the macro body, even if it cannot be a literal simple name because of illegal characters. For instance, if Z is 2006, and a macro is passed the first actual argument Y%dZ and the second actual argument,

the string `"foo"`, then `#1#%s#2#` is expanded as `Y2006foo`, even if `Z` is changed inside the macro.

2. One can see some similarities between composite names (with numeric suffixes) in Unimal and arrays in "normal" programming languages. In some cases, they are similar, but they are not the same. Likewise, there is some similarity between composite names (with string suffixes) in Unimal and associative arrays (or Perl-style hashes).

See also:

[Expand](#), [Simple names](#), [formats](#)

4.7. Target language interface

All lines of the input stream (i.e., original input file or any included files), which are not Unimal operators, are considered the lines of the target language.

Unimal copies target language lines to the output stream (i.e., standard output or the file specified in the last processed `Export` operator) while substituting the embedded target language interface items with their values printed according to the supplied format.

More precisely, Unimal searches for a signature

`#mp<format><name>`

where `<format>` is a valid Unimal format, and `<name>` is a simple or composite name, or

`#mp{<format><name>}`

When either signature is encountered, it is replaced with the value of the named macro parameter rendered according to the format.

In the curly braces case, white space characters are allowed anywhere between formats and components of a name for better readability.

Examples:

Assuming `S` has a string value `"ABCD"` and `X` has a numeric value `5` and `xABCD5` has a numeric value `17`,

`#mp%d%x%S%dX` is replaced by `17`, and

`#mp{%d x %s S %d X }U` is replaced by `17U`.

Note that the braced syntax is necessary if a Unimal target language interface statement does not end on a word boundary, as in the second example. In other words, a statement

`#mp%d%x%S%dXU`

will use the numeric value of the macro parameter `XU` (if available, otherwise it is an error). This is not what was intended

NOTE. Within a macro body, a formal argument is a valid simple name. During macro expansion, it is replaced with the value of the actual argument. As a word of caution, this means that if the second actual argument is a composite name, say, `a%ub` with a value `4`, then `#mp%ux%u#1#` means `#mp%ux4`, and *not* `#mp%ua%ub`.

See also:

[Formats](#)

4.8. Numeric Expressions

Expressions are used in some of the Unimal operators, such as `Set` or `If`. In their syntax, Unimal expressions are very similar to well-behaved integral expressions in the C programming language. **Unimal** syntax is more restrictive, though, to avoid confusing precedence rules of C.

4.8.1. Terms and operations

The simplest expressions are primary expressions, or *terms*. A term is

- A number literal, or
- A macro parameter (with simple or composite name) with a numeric value, or
- An actual argument in a macro expansion which resolves to a numeric value, or
- A result of evaluating a *numeric function* (numeric functions are covered in the Built-in Expressions section), or
- Any numeric expression enclosed in parentheses

Terms can be operands to *operations*. A result of an operation is a 32-bit signed integer number with 2's complement representation of negative numbers.

Unimal supports the following groups of operations described next:

- Arithmetic
- Shift
- Bitwise logic
- Comparisons
- Boolean logic

Operations within each group may have natural order of precedence (C-like, but not exactly). There is no precedence whatsoever between any two operations of different groups. A un-parenthesized mixture of disparate operations is a syntax error.

4.8.2. Arithmetic expressions

Arithmetic expressions are natural expressions with arithmetic operations on terms and arithmetic sub-expressions:

- `+` (Addition or unary `+`)
- `-` (Subtraction or unary `-`)
- `*` (Multiplication)
- `/` (Division)
- `%` (Remainder)

Unary operations take highest precedence, followed by multiplicative operations (`*`, `/`, `%`), followed by additive expressions (`+` and `-`).

Unary operations are right-associative, e.g., `- -x` is `-(-x)`.

All binary operations of the same precedence are left-associative, e.g., `x*y/u/v` is `((x*y)/u)/v`. For instance, `2/4*8` is the same as `(2/4)*8` and has a value 0 (recall that Unimal does integer arithmetic).

To change the order of operations, the sub-expression to be evaluated first must be enclosed in parentheses.

Parentheses can also be freely used to improve readability.

Unimal detects positive and negative overflow errors in addition, subtraction and multiplication. It also detects division by 0 (both on % and on /). As a way to reduce ambiguity, the remainder operation requires that the divisor be positive. The remainder is always a non-negative number

Note:

1. To avoid ambiguity, Unimal requires that the divisor in the Remainder (modulo) operation (%) be *positive*. The result is always non-negative in the range from 0 to (divisor - 1).

4.8.3. Shift expressions

Shift expressions are simple C-like shift operations on terms:

- << (Left shift)
- >> (Right shift)

E.g., a >> b indicates right shift of a by b counts (bit positions). If the shift count is negative, the result is the positive shift in the opposite direction by the absolute value of counts. E.g.

a >> (-5)

is the same as

a << 5.

Bits shifted in as a result of operation are always zeros. E.g. (-1)>>31 has a value 1 because negative numbers are represented in two's complement format (as is native to most computers).

In particular, the result of a shift by 32 or more counts is 0.

There is no precedence among shifts; any un-parenthesized mix is a syntax error.

4.8.4. Bitwise logic expressions

Bitwise logic expressions are a standard fare of C-like bitwise operations on terms and bitwise logic sub-expressions:

- ~ (Bitwise negation, or 1's complement)
- & (Bitwise AND)
- ^ (Bitwise exclusive OR)
- | (Bitwise OR)

Unary negation takes highest precedence, followed by &, followed by ^, followed by |.

Unary negation is right-associative, i.e., ~~x is a valid (but arguably useless) expression.

All binary expressions of the same precedence are left-associative, e.g., x&y&u&v is ((x&y) &u) &v.

To change the order of operations, the sub-expression to be evaluated out-of-order must be enclosed in parentheses.

Parentheses can also be freely used to improve readability.

4.8.5. Logical expressions

Logical expressions include:

- comparisons

- logical negation
- logical AND '&&'
- logical OR '||'
- built-in logical expressions (covered in the Built-in Expressions section)

4.8.5.1. Comparisons

Comparisons of operands *a* and *b* are

$a < b$ (less than)
 $a > b$ (greater than)
 $a == b$ (is equal to)
 $a != b$ (is not equal to)
 $a <= b$ (less than or equal to)
 $a >= b$ (greater than or equal to).

Valid operands for comparisons are:

- terms
- arithmetic expressions

E.g., $x+3 < 4$ is a valid expression but $x|3 < 4$ is not. (But then again, $(x|3) < 4$ is valid.)

The logical value of a comparison is a 1 if the comparison is true or 0 otherwise. Comparison operands, like operands of any other numeric operation, are signed numbers.

There is no precedence among comparisons; any mix is a syntax error.

4.8.5.2. Negation '!'

Negation operation is applicable to

- an atomic expression (including numeric and logical built-ins), or
- another negation

It has the highest precedence. Syntax:

`!<atomic_expression>`

It converts the expression's type to "logical expression" and makes the result a 1 if the expression evaluated to zero, and a 0 if the expression evaluated to non-zero.

Examples:

`!3` evaluates to 0
`!(3>5)` evaluates to 1
`!3>5` is a syntax error: there is no precedence resolution between '!' and '>'.

The negation operation is right associative, i.e., `!!x` is `!(!x)`.

4.8.5.3. Logical AND '&&' and OR '||' expressions

Basic logical expressions (negations and comparisons), as well as terms, can be combined in more complex expressions using logical AND and OR operations. Both are left-associative, e.g., `a&&b&&c` is `(a&&b)&&c`, and `&&` takes precedence over `||`, i.e., `a||b&&c` is `a|| (b&&c)`.

A (logical) value of a non-zero term in a logical expression is 1.

To change the order of operations, parentheses can be used.

Parentheses can also be freely used to improve readability.

4.9. String Expressions

A Unimal string expression is

- a string literal, or
- a macro parameter which holds a string value, or
- a built-in string expressions (covered in the Built-in expressions section)

A string literal has two forms.

One form is a quoted string, like `"foo"`; a literal quote must be doubled, e.g.

`""Quoted""`.

The other form is delimited by `#@` and `#`, like `#@foo#`; a literal `#` must be doubled, e.g.

`#@Train ##5#`.

4.10. Attributes

A compile-time variable can have one or more of the following attributes:

- `NUM`, if it holds a numeric value, and/or
- `STR`, if it holds a character string, and/or
- `MAC`, if it holds a macro definition

An *attribute list* is one or more of the attributes separated by commas and enclosed in curly braces, e.g. `{STR}` or `{NUM, MAC}`. Technically, attributes can be repeated in a list, as in `{NUM, NUM}` but the repeated entry has no effect.

4.11. Built-in Expressions

4.11.1. Logical built-ins

4.11.1.1. Defined

The syntax of the Defined expression is

```
Defined(<macro_parameter><attribute_list>)
```

The value of the expression is a 1 if the macro parameter has a value corresponding to *any* of the attributes in the list; it is a 0 otherwise.

Example:

`Defined (foo {MAC, NUM})` is a 1 if `foo` has a numeric value or if there is a macro defined with the name `foo`.

The attribute list may be omitted; in this case the default list `{NUM, STR}` is assumed.

4.11.1.2. Isconst

The syntax is

```
Isconst (<expression>)
```

The value of the expression is a 0 if the argument expression happens to be a macro parameter or a 1 otherwise (i.e. a literal or a non-trivial expression).

This built-in is only useful within a macro, such as when we want to know whether a value of `#3#`, the third argument, can be modified. This of course may change from one macro invocation to another.

Examples:

`Isconst("foo")` is a 1

`Isconst(foo)` is a 0 and is well-defined whether `foo` was ever assigned a value

4.11.2. Numeric functions

The common syntax of numeric functions is

`<function>(<argument list>)`

The argument list is a comma-separated list of expressions which must have types (attributes) according to the function.

4.11.2.1. Math functions

All math functions in Unimal take four numeric arguments.

A calculation of the Unimal `<function>(a, b, c, d)` is performed as the integer floor (largest integer no greater than) of the mathematical expression

$$\frac{a}{b} \cdot \langle function \rangle \left(\frac{c}{d} \right)$$

where a, b, c, d are floating-point representations of the integer arguments a, b, c, d respectively. Therefore, the second and the fourth arguments (b and d) must be non-zeros. Below is the table of supported math functions in Unimal:

Unimal function	Calculates	Math function	Valid arguments
<code>Usin(a, b, c, d)</code>	$\frac{a}{b} \cdot \sin\left(\pi \frac{c}{d}\right)$	sine	$-2 \leq \frac{c}{d} \leq 2$
<code>Ucos(a, b, c, d)</code>	$\frac{a}{b} \cdot \cos\left(\pi \frac{c}{d}\right)$	cosine	$-2 \leq \frac{c}{d} \leq 2$
<code>Uasin(a, b, c, d)</code>	$\frac{a}{b\pi} \cdot \arcsin\left(\frac{c}{d}\right)$	arcsine	$-1 \leq \frac{c}{d} \leq 1$
<code>Uacos(a, b, c, d)</code>	$\frac{a}{b\pi} \cdot \arccos\left(\frac{c}{d}\right)$	arccosine	$-1 \leq \frac{c}{d} \leq 1$
<code>Uatan(a, b, c, d)</code>	$\frac{a}{b\pi} \cdot \arctan\left(\frac{c}{d}\right)$	arctangent	
<code>Uexp(a, b, c, d)</code>	$\frac{a}{b} \cdot \exp\left(\frac{c}{d}\right)$	exponent	
<code>Ulog(a, b, c, d)</code>	$\frac{a}{b} \cdot \log\left(\frac{c}{d}\right)$	Natural logarithm	$\frac{c}{d} > 0$
<code>Usqrt(a, b, c, d)</code>	$\frac{a}{b} \cdot \sqrt{\frac{c}{d}}$	Square root	$\frac{c}{d} \geq 0$

If argument(s) do not meet the conditions in the "Valid arguments" column, Unimal generates an error. If the arguments are valid but the function evaluation causes overflow or underflow in floating point arithmetic, then Unimal tacitly uses a zero for both underflow and overflow. An error is generated on overflow, but not on underflow.

See also:

[Names](#), [numeric expressions](#)

4.11.2.2. Misc. functions

Currently, Unimal defines one "miscellaneous" function – `Ustrlen`. It takes one argument – a string expression – and returns a number equal to the length of the string *in one-byte characters*.

If your programmer's editor supports multi-byte characters (such as UTF8-encoded), you can enter them in a Unimal string literal, but `Ustrlen` will return the number of raw bytes in the string, not the number of encoded characters.

4.11.3. String expressions

Unimal string expressions have a common syntax of a comma-separated list of expressions in curly braces, where the first expression must be a named macro parameter:

```
{<name>, <expression>, ..., <expression>}
```

The `<name>` determines the expected number and types of expressions, and the actual operation performed on them.

4.11.3.1. Name-to-string conversion

Syntax:

```
{<name>}
```

Value:

A string containing the `<name>`

4.11.3.2. Substring extraction `uSubstr`

Syntax:

```
{uSubstr, <string>, <start>, <end>}
```

Arguments:

`<string>` - a string expression

`<start>` - ordinal number of the first character of the substring

`<end>` - ordinal number of the last character of the substring, minus 1

Value

A string starting with `<start>`-th and ending with the `(<end>-1)`-th character of the `<string>`. However, if `<start>` is greater than `<end>`, they are swapped and the characters in the substring appear in reverse order.

Character numbers are 0-based and extract nothing outside the string.

Example: Processing the following file:

```
#MP Setstr s = {uSubstr, "abcdefgh", 1, 100}
[#mp%ss]
#MP Setstr s = {uSubstr, "abcdefgh", 100, 1}
```

```

[#mp%ss]
#MP Setstr s = {uSubstr, "abcdefgh", 1, 7}
[#mp%ss]
#MP Setstr s = {uSubstr, "abcdefgh", 7, 1}
[#mp%ss]
#MP Setstr s = {uSubstr, "abcdefgh", 100, 200}
[#mp%ss]
#MP Setstr s = {uSubstr, "abcdefgh", 200, 100}
[#mp%ss]
#MP Setstr s = {uSubstr, "abcdefgh", -4, 4}
[#mp%ss]
#MP Setstr s = {uSubstr, "abcdefgh", 4, -4}
[#mp%ss]
#MP Setstr s = {uSubstr, "abcdefgh", 4, 4}
[#mp%ss]

```

gives this output

```

[bcdefgh]
[hgfedcb]
[bcdefg]
[gfedcb]
[]
[]
[abcd]
[dcba]
[]

```

4.11.3.3. Concatenation uJoin

Syntax:

```
{uJoin, <string>, ..., <string>}
```

Value:

Concatenation of all strings in the list. If, however, a macro parameter uJoin has a string value, this value is inserted between any two consecutive <string>s.

Example: Processing the following file:

```

#MP Setstr s = {uJoin, "This", "made", "my", "day"}
[#mp%ss]
#MP Setstr uJoin = " "
#MP Setstr s = {uJoin, "This", "made", "my", "day"}
[#mp%ss]

```

gives this output

```

[Thismademyday]
[This made my day]

```

4.11.3.4. Simplified Concatenation

Syntax:

```
<string> <separator> <string> <separator> ... <string>
```

A <separator> can be a space or a tab character or a + sign, with any number of additional spaces or tabs around.

This syntax is valid only in Setstr operator.

Value:

Concatenation of all strings in the list.

4.11.3.5. Splitting a string uSplit

Syntax:

```
{uSplit, <string>, <chop string>, ..., <chop string>}
```

Value and side-effects:

This expression finds the first occurrence of any of the <chop string>s in the <string>. If more than one <chop string> matches at the same place, the first longest match is taken.

The macro parameter uSplit is given a numeric value – the 0-based number of the matching <chop string>.

If <string> is itself a macro parameter, it, too, gets a numeric value – the position in the <string> just beyond the match.

The value of the expression is the segment of the <string> before the match.

If no match is found, uSplit gets a negative value and the value of the expression is the whole <string>.

Caution: an empty string ("") matches at the beginning of the <string> and the return value is an empty string.

Example: Processing the following file:

```
#MP Setstr s = #@qwertyuiop#
#MP Setstr s1 = {uSplit, s, #@ty#, #@tyu#}
Segment is #mp%ss1
Best-match string is #mp%duSplit
Remainder is at #mp%ds
#MP Setstr r = {uSubstr, s, s, 1000}
Its value is #mp%sr.
```

gives this output

```
Segment is qwer
Best-match string is 1
Remainder is at 7
Its value is iop.
```

4.11.3.6. Defined encoding

Syntax

```
[<numeric expression>]
```

Value:

A single-character string with the character encoding being the least significant byte of the <numeric expression>

Example:

```
#MP x=0x10B
#MP Setstr y = [x*x]
```

x*x is 0x11679 and its least significant byte is 0x79. So y is a string whose single character is encoded as 0x79. If the character encoding happens to be ASCII, y has a value "y".

This feature is useful in creating and manipulating “texts” displayed on the devices with fixed encoding.

4.12. Unimal Operators

Unimal operators, or statements, are identified by the signature `#MP` in the beginning of the line (after optional leading white spaces). They control how the output stream is generated but do not produce any output by themselves.

It is a general rule that white spaces are optional (unless necessary as separators) and can be added freely as desired. In particular, a white space between the signature `#MP` and the operator keyword is optional.

All Unimal operators are line-based. There is no line continuation feature in Unimal operators.

All characters of a Unimal statement line past the semicolon `;` are considered comments and ignored by Unimal.

Unimal offers a standard fare of block operators (loops and conditional executions; that is to say, rescanning a segment of input and conditional expansions) and an advanced macro facilities. A unique feature is that a macro may contain a partial block (e.g. beginning of a loop). A constraint by design is that any input file (whether the file supplied to Unimal or a directly or indirectly included file) must have, after all macros are expanded, only complete blocks, if any. That is, it cannot have e.g. a dangling `If` for which there is `Endif` or vice versa.

Here is an example (necessarily with forward references):

Foo.u

```
#MP Macro foo
#MP If 0
#MP Endm
```

Bar.u:

```
#MP Include "Foo.u"
#MP Expand foo
#MP ;Error 2000 - unmatched If after macro expansion
```

Baz.u

```
#MP Include "Foo.u"
#MP Expand foo
#MP Endif ;closes the If block opened by the macro expansion
#MP ;OK
```

4.12.1. Empty operator

Syntax:

```
#MP
```

Description

The empty operator doesn't do anything. Properly indented, it comes handy as a line spacer not copied to the output stream. It is also useful for long single-line comments of Unimal code.

4.12.2. For

Syntax:

```
#MP For <name>=<expression1>, <expression2>
```

Description

The **For** operator defines a macro parameter <name> (unless already defined), assigns it the value of numeric <expression1>, captures the current value of numeric <expression2> and starts the **For** loop block terminated by the matching **Endfor**. The block is processed repeatedly zero or more times for every value of the macro parameter <name> from the initial value and to the initially captured value of the <expression2> inclusive. The loop counter <name> is incremented by 1 automatically when the **Endfor** is processed. The loop counter can be manipulated within the loop body, in which case the number of times the body is processed is manually controlled.

See also:

[Endfor](#), [numeric expressions](#), [names](#)

4.12.3. Endfor

Syntax:

```
#MP Endfor
```

Description

Ends the **For** loop block started by the matching **For** operator (in the same level of operator nesting). The block is processed repeatedly zero or more times for every value of the macro parameter <name> from the initial value equal to the <expression1> and to the previously captured value of the <expression2> inclusive. The loop counter <name> is incremented by 1 automatically when the **Endfor** is processed. The value of the loop counter upon exit from the loop is one greater than the <expression2> in the matching **For** operator, unless

- The loop executed zero times. In this case, the loop counter remains equal to the initial value (<expression1> in the matching **For** operator).
- The loop is modified manually within the loop body. In this case, it may be the last assigned value.

See also:

[For](#)

4.12.4. Repeat

Syntax:

```
#MP Repeat
```

Description

This operator marks the beginning of the **Repeat/While** loop block which ends with a matching **While** operator. The block is processed repeatedly one or more times depending on the decision made by the matching **While**.

See also:

[While](#)

4.12.5. While

Syntax:

```
#MP While <expression>
```

Description

The `While` operator marks the end of a `Repeat/While` loop. It evaluates the numeric `<expression>`, and if the result is non-zero, continues processing from the matching `Repeat` statement.

See also:

[Repeat](#), [numeric expressions](#), [names](#)

4.12.6. If

Syntax:

```
#MP If <numeric expression>
```

Description

Begins an `If/Else/Endif` block. If the `<expression>` evaluates to non-zero, the block between `If` and `Endif` or optional `Else` is processed, otherwise, it is skipped.

See also:

[Else](#), [Endif](#), [numeric expressions](#)

4.12.7. Else

Syntax:

```
#MP Else
```

Description

Begins an optional `Else` part of an `If/Else/Endif` block of the same level of operator nesting. If the corresponding `If` or `Ifdef` operator evaluated to true condition, the block between `Else` and `Endif` is skipped; otherwise, it is processed.

See also:

[If](#), [Ifdef](#), [Endif](#)

4.12.8. Endif

Syntax:

```
#MP Endif
```

Description

Ends an `If/Else/Endif` block of the same level of operator nesting.

See also:

[Else](#), [If](#), [Ifdef](#)

4.12.9. Ifdef

Syntax:

```
#MP Ifdef <name> <attributes list>
```

Description

This is simply shorthand for

```
#MP If Defined(<name> <attributes list>)
```

The attributes list may be omitted; in this case the default list {NUM, STR} is assumed.

See also:

[Else](#), [Endif](#), [names](#)

4.12.10. Set

Syntax:

```
#MP Set <name>=<expression>
#MP Compute <name>=<expression>
#MP Set <name> <expression>
#MP Compute <name> <expression>
#MP <name>=<expression>
```

Description

Defines the macro parameter <name> (if not already defined) and assigns it the value of the numeric <expression>. All forms are equivalent. [Set](#) and [Compute](#) are synonyms.

Note that either [Set](#) ([Compute](#)) or the equal sign can be omitted, but not both.

See also:

[Names](#), [numeric expressions](#)

4.12.11. Setstr

Syntax:

```
#MP Setstr <name>=<string expression>
#MP Setstr <name> <string expression>
```

Description

Defines the macro parameter <name> (if not already defined) and assigns it the value of the <string expression>. Note that the equal sign may be omitted.

See also:

[Names](#), [expressions](#)

4.12.12. Macro

Syntax:

```
#MP Macro <name>
```

Description

Starts a definition of a macro, which ends with the [Endm](#) operator. Valid only outside of any other macro definition (in other words, macro definitions cannot be nested). The content of the macro definition block is completely ignored until the macro is expanded (see [Expand](#)). Within a macro body, special rules apply to Unimal names.

A macro definition cannot span across different files (such as the beginning in one include file, and the end, in another).

Note: A macro definition placed inside an [If/Else](#) or a [For](#) block is noticed only when the block is processed (i.e., when the block condition is true).

See also:

Names, [Endm](#), [Expand](#)

4.12.13. Endm

Syntax:

```
#MP Endm
```

Description

Ends a definition of a macro which begins with the [Macro](#) operator.

See also:

[Names](#), [Macro](#), [Expand](#)

4.12.14. Expand (non-recursive)

Syntax:

```
#MP Expand <name> (<arguments>)
#MP <name> (<arguments>)
#MP Expand <name>
#MP <name>
```

Description

`<name>` is an already defined name of a macro or a string expression evaluating to a name of an already defined macro. `<arguments>` is a comma-separated list of arguments, which can also be empty (and if it is empty, parentheses may be omitted). The keyword [Expand](#) is optional and may be omitted a matter of style.

Syntactically, an argument can be any valid Unimal expression.

The [Expand](#) operator replaces itself with a literal expansion of (a copy of) the body of the named macro as follows:

First, all actual arguments are evaluated.

If an argument is a macro parameter, its (composite) name is resolved to a simple name using the values of other parameters currently in effect. E.g., assuming `bar` is 5 and `baz` is "dude" `foo%dbar%sbaz` is replaced with `foo5dude`. (Simple names therefore remain unchanged.)

If, on the other hand, an argument is a numeric or a string expression, the expression is evaluated using the current values of its operands.

For example, the two [Expand](#) operators below are equivalent

```
#MP narg = 5
#MP Setstr sarg = "the argument"
#MP Expand foo(narg, sarg%dnarg, narg+1, {sarg})
#MP foo(narg, sarg5, 6, "sarg")
```

Second, the [Expand](#) operator substitutes any formal arguments within the macro body with the corresponding (resolved) actual arguments from the argument list. Therefore, the type and the number of actual arguments must match the use of formal arguments in the macro body. Any mismatch is likely to cause a syntax error which is occasionally

difficult to trace. On rare occasions, a mismatch may produce a syntactically correct expansion with unexpected values.

Notes:

1. Operator blocks (`If/Else/Endif` or `For/Endfor`) can span different macros, perhaps, in different levels of macro nesting (or be partly outside of any macros).
2. A macro parameter, such as `sarg5` in the example above, doesn't have to be defined before being passed as an argument to a macro.
3. A macro expansion cannot be recursive (directly or indirectly), i.e. an `Expand` operator may not occur in the expansion of the macro with the same name.

See also:

[Names](#), [Numeric expressions](#), [Strings](#), [Endm](#), [Macro](#), [Target language interface](#)

4.12.15. Expand (possibly recursive)

Syntax:

```
#MP Expand <name>[<arguments>]
#MP <name>[<arguments>]
```

Description

This form of the `Expand` operator syntactically differs from a non-recursive form above in that the argument list is passed in square brackets instead of parentheses, and an empty argument list cannot be omitted.

In a normal (non-recursive) form of the `Expand` operator, a macro must be expanded normally even if the operator is in a false block of an `If` operator because the expansion may yield e.g. an `Endif` for that block buried perhaps in some contained macro.

The result of this is that any recursive expansion of a macro would automatically produce an endless recursion. Unimal detects an `Expand` operator what would cause a recursion, generates Error [S2022](#), and skips the expansion.

To instruct Unimal that there is no unbalanced block elements in the macro to be expanded (including nested macros), you must use the bracketed variant of the `Expand` operator. With it, Unimal skips the expansion in a false block and performs a perhaps recursive expansion in a true block.

With the exception of the recursion and false block treatment, the bracketed variant of the `Expand` operator is equivalent to the normal parenthesized variant. However, it may be worth noting that this bracketed variant of the `Expand` operator is slightly faster.

4.12.16. Include

Syntax:

```
#MP Include <string expression>
```

Description

The `Include` operator switches the input stream from the current file to the file specified in the `<string expression>`. It is also fair to say that `Include` literally expands the whole include file in its place.

The interpretation of the filename depends on the operating system, and what is a valid filename on one platform can be invalid on another. See the section Invoking Unimal on how the include file is searched.

See also:

[Strings](#), [Macro](#)

4.12.17. Export

Syntax:

```
#MP Export (numeric expression) <string expression>
#MP Export Push
#MP Export Pop
```

Description

The first form of the [Export](#) operator switches the output stream from the current file to the file specified in the <string expression>.

The `numeric expression` must be enclosed in parentheses. If it evaluates to non-zero, then the Unimal output is appended to the existing file (if it already exists). Otherwise, the output file is overwritten from the beginning.

NOTE. Redefining the current output file with the same file (with the zero-valued expression) causes the previous output to this file to be lost.

The interpretation of the filename depends on the operating system, and what is a valid filename on one platform can be invalid on another. See the section Invoking Unimal on where the output file is placed.

Special case: When the <string> is empty (i.e., is literally equal to "" or #@#), the output stream is sent to the standard output, which is the default. On the command line level, you can redirect standard output to a file of your choice.

The new output file remains in effect until redefined by another [Export](#) operator.

The second form of the [Export](#) operator saves, and the third form restores the output file information. Presumably, the output file is pushed, then replaced with some other file and then popped to continue output to it. Export Pushes/Pops can be nested.

See also:

[Strings](#)

4.12.18. End

Syntax:

```
#MP End
```

Description

The [End](#) operator is a common synonym of [Endfor](#) and [Endif](#): Whenever you'd use [Endfor](#) or [Endif](#), you can use [End](#).

(In fact, the `End` operator is primary, and `Endfor` and `Endif` are its synonyms. This means that it is legal to close the `For` block with `Endif` and the `If` block, with `Endfor`. Obviously, it is not a good idea.)

See also:

[Endfor](#), [Endif](#)

4.12.19. Undef

Syntax:

```
#MP Undef <name> <attributes list>
```

Description

The `Undef` operator undefines a name (in the “namespace” of each attribute in the list). If the name was not previously defined, the `Undef` has no effect.

The `<attributes list>` may be omitted; in this case the default list `{NUM, STR}` applies.

Example

To ensure a macro definition does not collide with a previous definition, we can go brutal:

```
#MP Undef foo{MAC}  
#MP Macro foo  
.....
```

See also:

[Set](#), [Setstr](#), [For](#)

4.12.20. Save

Syntax:

```
#MP Save <name>
```

Description

The `Save` operator saves all defined values of the named parameter. Its purpose is to restore the saved values with `Restore`.

See also:

[Restore](#)

4.12.21. Restore

Syntax:

```
#MP Restore <name> <attributes list>
```

Description

The `Restore` operator restores the named parameter’s values with the given attributes.

If the previously `Save`'d value with some attribute was not defined, any current definition is lost: in this case `Restore` acts like `Undef`.

If the name was not previously `Save`'d, it is treated as saved with all undefined values, and the `Restore` acts exactly like `Undef` with the same attributes.

The `<attributes list>` may be omitted; in this case all saved values `{NUM, STR, MAC}` are restored.

Examples

The following shows what happens with the definitions of previously undefined `foo`:

```
#MP foo = 9
#MP Restore foo {MAC} ;foo (numeric) is 9
#MP Save foo
#MP Undef foo
#MP Setstr foo "hello"
#MP Restore foo ;string value is lost, numeric value is 9
```

The following temporarily redefines a macro `foo`:

```
#MP Save foo
#MP Macro foo
<new definition>
#MP Endm
<do something with the new version of foo>
#MP Restore foo {MAC}
<do something with the original version of foo>
```

See also:

[Save](#), [Undef](#), [Attributes](#)

4.13. A useful shorthand for a list of arguments

It is not unusual within a macro expansion to pass a number of consecutive arguments to a nested macro. Unimal offers a shorthand for it, in the form `[m:n]`, where `m` and `n` are numeric expressions. E.g., to invoke some macro `FOO` with parameters 3, 5, all the arguments of the current macro, and 6, we can write:

```
#MP Expand FOO(3, 5, [1 : #0#], 6)
```

The `[m:n]` syntax can be used where a comma-separated list of parameters is used, e.g., in string expressions.

Outside of a macro expansion, `[m:n]` always produces an empty sub-list of arguments regardless of (valid) values of `m` and `n`. Inside a macro expansion, if `m < 1` it is replaced with 1, and if `n > #0#`, it is replaced with `#0#`, the total number of actual parameters in the current macro expansion. If `m > n`, the sub-list is empty.

Here is a simple example where this syntax is useful. Consider a macro `FOO` which invokes one of the other macros (`FOO1`, `FOO2` ...) depending on the value of the first argument, and passes the rest of the arguments for processing. The idea is to put in the definition of `FOO` something like

```
#MP Expand FOO%d#1# (?)
```

But – what goes in place of the question mark?

Without the shorthand, we had to write:

```
#MP If #0#==1
#MP Expand FOO%d#1# ()
#MP Endif
#MP If #0#==2
#MP Expand FOO%d#1# (#2#)
#MP Endif
#MP If #0#==3
#MP Expand FOO%d#1# (#2#, #3#)
#MP Endif
```

... And so on. That's cumbersome and not maintainable. With the shorthand, we can write simply

```
#MP Expand FOO%d#1# ([2:#0#])
```

4.14. Special macro parameters

There are several macro parameters which Unimal treats in special ways:

- `uErrorFormat`
- `uAutoLine`
- `uAutoLineOut`

`uErrorFormat`, if it has a string value, changes the default format of Unimal error messages; see Section [Error message format mimicry](#).

Description of the other specially named macro parameters follows.

4.14.1. `uAutoLine`

Unimal automatically maintains the file name and the line number currently being processed. They are accessible as the string value and as the numeric value respectively of the macro parameter named `uAutoLine`.

If the *macro* `uAutoLine` is defined (by the programmer), Unimal automatically expands it whenever line numberings of the input and of the output diverge. During automatic expansion of the `uAutoLine` macro, including expansion of its nested macros (if any), both numeric and string values of `uAutoLine` are frozen.

This seemingly odd feature is designed to provide a flexible method of emitting input file/line information into the output stream, such as e.g., `#line` directives in C-like languages.

Assuming for an example that the target language is C, the programmer could put a target language interface line

```
#line #mp%uAutoLine "#mp%suAutoLine"
```

anywhere she likes to ensure the synchronization.

If, however, it is put in the `uAutoLine` macro:

```
#MP Macro uAutoLine
```

```
#line #mp%uuAutoLine "#mp%suAutoLine"  
#MP Endm
```

then the synchronization `#line` statements will be emitted automatically as needed.

Notes:

1. Notice that the macro `uAutoLine` takes no arguments.
2. Writing to `uAutoLine`, like

```
#MP Setstr uAutoLine = "foo"
```

does not produce an error but is evil and will be overwritten by Unimal.
3. While a means of automatically emitting the file/line information was a primary motivation for automatic expansion of the `uAutoLine` macro, there is no limitation on what exactly this macro does.
4. It is legal to expand `uAutoLine` normally, like

```
#MP Expand uAutoLine()
```

4.14.2. `uAutoLineOut`

Unimal automatically maintains the file name and the line number currently being output. They are accessible as the string value and as the numeric value respectively of the macro parameter named `uAutoLineOut`.

The primary purpose of the macro parameter `uAutoLineOut` is to resynchronize to the output file itself after previous synchronization to the input file, e.g.,

```
#MP Undef uAutoLine {MAC}  
#line #mp%uuAutoLineOut "#mp%suAutoLineOut"
```

If you need to switch back and forth between synchronization to input and to output, a better solution is to make the macro `uAutoLine` emit one or the other synchronization directive (or none at all). This issue may be covered in greater detail in an application note.

The string value of `uAutoLineOut` is updated only when the output filename changes, so writing to it is even more evil than writing to `uAutoLine`.

5. Error Detection and Recovery

5.1. Error logging mechanism in Unimal

When **Unimal** detects an error, it uses the corresponding error message for two purposes.

The first purpose is obvious: to point the programmer to the offending Unimal statement and to assist with diagnosing the problem.

The second purpose is slightly less obvious. It is to mangle the output file so it does not compile as a source code in the target language.

These two purposes somewhat contradict each other: since Unimal can generate multiple output files (switched by the Export operator), searching for error messages can be troublesome. Besides, if an error occurred in accessing the output file, it would be lost if not duplicated elsewhere.

To counter these problems, Unimal logs identical error messages in the current output file and in the file with the fixed name "unimal.err." On top of that, the error is also sent to the standard error device (`stderr`, normally, the console terminal).

This way, no error gets lost, the erroneous output files won't compile, and the programmer has all error messages conveniently collected in "unimal.err."

The error file is generated in current working directory. It is overwritten on each run of Unimal. To avoid possible confusion, the first line of the file indicates which source file was processed. The last line of the file contains the number of errors encountered. Here is a sample content of `unimal.err`:

```
Processing C:\Projects\Unimal\sample.u ...
```

```
No errors
```

Unimal outputs all errors in the order of detection. Even though some errors are induced by a previous error, the error output allows better understanding of the root cause of the failed execution.

Since the Unimal language is line-based, Unimal resynchronizes from an error when it reaches the end of the line.

This is not to say, of course, that Unimal's error detection is free from induced errors. For instance, if a `For` operator has an error, then there will probably be unmatched `Endfor` and numerous instances of undefined loop counter used within the `For` block.

5.2. Unimal error messages reference

5.2.1. Default format of an error message

Here is an example of a typical Unimal error message:

```
MP:S2001:remove.u:1 Bad syntax near Incl; statement ignored
```


An error message always starts with the signature "MP:" followed by:

- A single-letter error type (in our example, 'S') and
- A four-digit error number,
- The current input file name (in our example, file is `remove.u`) and
- Line number (in our example, the line number is 1).

This information is followed by a textual description of the error, including, when applicable, the offending element (`Incl` in our example) and the action taken by Unimal.

Unimal can take one of the three kinds of actions upon encountering an error:

- Abort processing immediately
- Give up and just ignore the offending statement altogether
- Accept part of a wrong input as a valid statement and ignore the remaining part of it.

A typical case of the latter is a forgotten semicolon before a comment in a Unimal operator. There are much less innocuous errors of this kind, though.

Below (beginning with Section 5.5.3) is a description of Unimal errors by their type, in default format.

5.2.2. Error message format mimicry

In some instances it is desirable to have error format readily understood by an existing error parsing mechanism. A typical case is an IDE supplied with your compiler; clicking on an error line in the "build output" window takes you to the file and line where an error occurred.

To make integration of Unimal with your development tool easier, Unimal allows to redefine the error output format. Namely, if a macro parameter `uErrorFormat` has a string value, this string is considered the error format string. An error message is the output format string with the following replacements:

- `$$` is replaced with `$`
- `$B` is replaced with line break (newline)
- `$F` is replaced with the input file
- `$L` is replaced with the input line number
- `$C` is replaced with the error class
- `$N` is replaced with the error number
- `$M` is replaced with a descriptive message
- `$<any-other-character>` is no character at all.

For instance, default error format can be thought of as

`"$BMP:$C$N:$F:$L $M"`.

As a different example, Microsoft C/C++ compiler error format style is achieved with

`"$F($L) : error CN: $M"`.

Please note that it is possible to set the error format string to something very non-descriptive, like `"Oops!"` or even an empty string `""`. Unimal doesn't (and cannot) check the sensibility of a user-defined error format; you use custom error format it is your responsibility to make it meaningful.

5.2.3. F type: Fatal file errors

If Unimal encounters any file-related error, it aborts processing immediately and exits.

This includes not only physical medium errors but also file system errors, such as non-existing file or too many open files.

If applicable, the corresponding system error message is appended in parentheses, such as "(No such file or directory)", but be aware that this message can be somewhat cryptic.

5.2.3.1. Error 0102

Message:

`No pushed output file to pop`

Reason:

Unimal cannot restore output because no output stream was saved.

Action by Unimal:

Continue output to the current output stream send error message to STDERR.

5.2.3.2. Error 0103

Message:

`Error reading input <file_name>; aborting`

Reason:

Error opening or reading the input file

Action by Unimal:

Abort processing; send error message to STDERR.

5.2.3.3. Errors 0104, 0105 (output file), 0106 (input file)

Message 0104:

`Can't open output <file_name>; aborting`

Message 0105:

`Error writing output <file_name>; aborting`

Message 0106:

`Can't access input <file_name>; aborting`

Reason:

Unimal cannot write to the specified output file or read from the specified input file. For instance, the file is locked by another application or the directory doesn't exist. Alas, you need to look at the system error message appended.

Action by Unimal:

Abort processing; send error message to STDERR.

5.2.3.4. Error 0111

Message

`Error processing command-line option <offending text>`

Reason:

Unimal failed to understand a command line argument.

Action by Unimal:

Abort processing; send error message to STDERR.

5.2.4. Special F type error (Usage syntax)

Message 0099:

<Short usage help text>

Reason:

Unimal was invoked with incorrect command line arguments.

Action by Unimal:

Abort processing; send error message to STDERR.

5.2.5. A type: Out of memory

If the computer runs very low on resources, or if your Unimal code uses tons and tons of names, macros etc., this type of errors can occur. This is a highly unlikely event.

The error message doesn't follow a standard scheme; it is simply

Out of memory
printed to STDERR.

Action by Unimal:

Abort processing.

5.2.6. S type: Syntax errors

5.2.6.1. Error 2000 (The file has an unbalanced beginning or end of a block)

Message 2000:

Unbalanced (missing) Endfor/While/Endif/Endm at end of file
Unbalanced (extra) Endfor/While/Endif/Endm at end of file

Reason:

First form: A **For** or a **Repeat** or an **If (Ifdef)** block is not closed at end-of-file, or a macro definition is not closed.

Second form: An **Endfor**, or a **While**, or an **Endif** was found that doesn't have a respective **For**, **Repeat**, or **If (Ifdef)** expanded while processing the current file. In other words, any block (after any macro expansions) must start and end within the same input file.

Action by Unimal:

Abort processing.

5.2.6.2. Error 2001 (General syntax error)

Message:

Bad syntax near <text>

This message may be augmented with a more detailed description.

Reason:

A syntax error in the statement; see important discussion below.

Action by Unimal:

If Unimal recognized a part of the text as a valid statement then the recognized part is processed and the remainder of the line is ignored. If Unimal didn't recognize a valid statement (first message), the whole line is ignored. That is, Unimal makes a lame attempt to recognize the programmer's intent. But even so, induced errors are not unlikely.

Common syntax errors

Syntax errors can be as simple as a misspelled operator keyword or forgotten semicolon before the comment, or they may have other causes, in which case the error message may look rather cryptic and confusing.

Essentially, any error in a statement can cause this error. Here are the most common:

Error	Likely <text>
Parentheses forgotten in an operator or an expression	<<EOL>> (end-of-line) or `;'`
Comma between arguments forgotten	The first lexical element of the next argument or `)`
Wrong type of a parameter (see below the special discussion on formal/actual arguments in macro expansions)	Whenever it is found, e.g., in Set 1+X=X it would be a 1, and in Set X+1=X it would be a `+'.
Expression syntax	The first non-blank element after the largest recognized part of the expression, e.g., in Set X=1+1++2 it would be a `+'.

Syntax errors in expressions

The second common case of an error is most likely to be induced either by a lexical error, or by a syntax error in an expression. For instance, an operator

```
#MPSet x = 1<1&1
```

would produce the message

```
Bad syntax near &; text ignored.
```

Special case: syntax errors in macro expansions

An offending token that is reported in an error message could be a macro's formal argument, such as `#1#`. While it looks confusing, the reason is the wrong argument type. For instance, consider this statement in a macro definition:

```
#MPSet #1# = X%d#2#
```

This is a perfectly good statement by itself, but it does assume the first actual argument to be a macro parameter (to which a value can be assigned), and not an expression. Similarly, it assumes the second actual argument to have a numeric value, e.g., to be a macro parameter with a numeric value or a numeric expression.

Therefore, when an actual argument 1 is, say, the number 17, and the actual argument 2 is the string, say, `#@ mystring#`, then the statement above would expand to

```
#MPSet 17 = X%d#@ mystring#
```

Clearly, it is syntactically wrong on both sides of the equal sign, so an error is reported. Again, there is nothing syntactically wrong in the macro definition, but the substitution of the formal arguments with actual arguments of unexpected types produced errors. You may need to inspect the corresponding `Expand` operator to trace the problem.

5.2.6.3. Error 2002 (Macro redefinition)

Message:

```
Macro <macro_name> redefinition; ignored
```

Reason:

A macro with this name has already been previously defined or the name has been previously used for a macro parameter.

Action by Unimal:

Unimal ignores the redefinition and sticks to the original definition of the named macro.

Note: It is *not* an error to encounter the *same* macro definition several times (such as when a file with the definition is included more than once or when, for some reason, a macro is defined within a `For` loop).

5.2.6.4. Error 2004 (Missing actual argument)

Message:

Missing actual macro arg <number>; 0 assumed

Reason:

A `#<number>#` term was encountered in a macro body during macro expansion where <number> is greater than the number of actual arguments.

Action by Unimal:

Unimal assumes the missing argument to be the number 0. In some cases, especially when the type guess was wrong, it induces other errors.

5.2.6.5. Errors 2005, 2006, 2007 (Unmatched block operators)

Message 2005:

Unmatched Else; ignored

Message 2006:

Unmatched Endfor/Endif; ignored

Message 2007:

Unmatched Endm; ignored

Reason:

One of the referenced operators was encountered when the corresponding preceding `If/Ifdef`, `For` or `Macro` operator was not encountered before.

Action by Unimal:

Unimal ignores the unmatched operator.

5.2.6.6. Error 2009 (Bad macro reference)

Message:

Undefined macro <macro_name>; ignored

Reason:

The `Expand` operator references a name that was not previously defined as a macro name (by a `Macro` operator).

Action by Unimal:

Unimal ignores the `Expand` operator.

5.2.6.7. Error 2010 (Unexpected type)

Message 2010:

Expected numeric value; default assumed

Expected string value; default assumed

Reason:

An operand in an expression or a statement was expected to have a numeric or a string value respectively, but it didn't.

Action by Unimal:

A default numeric value (0) or a default (empty) string is used.

5.2.6.8. Error 2011 (undefined parameter)**Message:**

Undefined parameter <parameter_name>; default assumed

Reason:

The macro parameter name was not defined in the "namespace" of the expected type.

Action by Unimal:

Unimal assumes a zero numeric or an empty string value.

5.2.6.9. Error 2012 (formatting in composite names or target language interface)**Message:**

Rendering format <format> incompatible with suffix type

Reason:

The macro parameter (or an actual macro argument) specified with the <format> does not match the type.

Action by Unimal:

Unimal replaces invalid formatting with *****.

5.2.6.10. Error 2013 (expected macro parameter)**Message:**

Named parameter expected in this context

Reason:

An expression or a literal found in the context where a named macro parameter was expected

Action by Unimal:

Unimal uses an empty name.

5.2.6.11. Error 2014 (expected a numeric)**Message:**

Non-numeric unexpected; 0 used

Reason:

A construct that doesn't resolve to a number is found where a numeric value was expected

Action by Unimal:

Unimal uses a 0.

5.2.6.12. Error 2015 (undefined string expression)**Message:**

String operation <name> not defined; name used

Reason:

In a string expression there are two or more arguments, and the first argument is not one of the pre-defined names.

Action by Unimal:

Unimal uses the name of the first argument as the string value of the expression.

5.2.6.13. Error 2016 (invalid string expression)**Message:**

No such string operation: "" used

Reason:

In a string expression there are no arguments, or the first argument is not a named macro parameter.

Action by Unimal:

Unimal uses an empty string as the string value of the expression.

5.2.6.14. Error 2017 (wrong number of arguments to a function)

Message:

`Wrong number of arguments to the <name> function; default result assumed`

Reason:

In a string expression or in a numeric function, the number of arguments is invalid.

Action by Unimal:

Unimal uses an empty string or a 0 as the value of the expression.

5.2.6.15. Error 2018 (literal number too large)

Message:

`Literal number <number> too large; replaced with maximum`

Reason:

A very large number, greater than 2,147,483,647, is used as a numeric literal.

Action by Unimal:

Unimal uses 2,147,483,647 as the value of the literal.

5.2.6.16. Error 2019 (string not terminated)

Message:

`Literal string [<string>] not closed`

Reason:

Probably, a terminating character (a quote or a sharp sign) is missing by the end of the line.

Action by Unimal:

Unimal uses <string> it collected to the end of the line as the value of the literal string.

5.2.6.17. Error 2020 (Nested macro definition)

Message:

`Nested macro definition; ignored`

Reason:

A `Macro` operator was encountered within another macro body.

Action by Unimal:

Unimal ignores the following macro definition up to and including the matching `Endm`.

Note: A nested macro definition complete with `Endm` should produce an induced error 2007, "Unmatched Endm".

5.2.6.18. Error 2021 (Unmatched While)

Message:

`Unmatched While; ignored`

Reason:

A `While` operator was encountered without the matching `Repeat`.

Action by Unimal:

Unimal ignores the unmatched operator.

5.2.6.19. Error 2022 (Recursive macro expansion)

Message:

```
Recursive use of macro <name>; ignored (use [])
```

Reason:

A () form of **Expand** operator was encountered such that a recursive expansion would occur.

Action by Unimal:

Unimal doesn't expand the macro.

5.2.7. L type: Lexical errors

If a sequence of characters cannot be interpreted as a valid element of the language in a Unimal operator, then a lexical error 4000 is reported along with the offending sequence of characters.

Message 4000:

```
Unrecognized text "<offending_text>"; ignored
```

Reason:

Odd characters in the input stream

Action by Unimal:

Unimal ignores the text that caused the error.

5.2.8. M type: Math errors

5.2.8.1. Errors 3500, 3501, 3502 (Arithmetic overflows)

Message 3500:

```
Addition overflow; result <value> assumed
```

Message 3501:

```
Subtraction overflow; result <value> assumed
```

Message 3502:

```
Multiplication overflow; result <value> assumed
```

Reason:

An overflow occurred in a respective operation in an expression.

Action by Unimal:

Unimal assumes the closest valid number in the direction of overflow.

5.2.8.2. Error 3503 (Divide by zero)

Message:

```
Divide by 0; result 0 assumed
```

Reason:

An attempt of division by zero is encountered.

Action by Unimal:

Unimal assumes zero result.

5.2.8.3. Error 3504 (Non-positive divisor in remainder operation)

Message:

```
Remainder divisor 0 or negative; result 0 assumed
```

Reason:

An attempt of remainder (%) division by zero or a negative number is encountered.

Action by Unimal:

Unimal assumes zero result.

5.2.8.4. Errors 3510, 3511, 3512, 3513 (math functions errors)

Message 3510:

```
Unknown function <function_name>; result 0 assumed
```


Reason:

The named function is not a valid name of a Unimal math function.

Message 3511:

Zero denominator; result 0 assumed

Reason:

The second or the fourth argument (a denominator in a ratio) of a Unimal function is zero.

Message 3512:

Math argument out of range; result 0 assumed

Reason:

The arguments to the Unimal function are outside of the range specified for that function.

Message 3513:

Math overflow; result 0 assumed

Reason:

The mathematical value of the function is outside the range allowed for 32-bit numbers.

Action by Unimal:

In all these cases, Unimal assumes zero result.

5.2.9. Internal errors

There are a few safeguard error messages coded into Unimal to trap unexpected problems. The corresponding errors should never be encountered. If, however, you see a suspicious error message not described above, please, contact MacroExpressions for the problem resolution.

6. Additional facts

6.1. (No) implementation limits

Unimal 2.0 (and higher) does not have implementation-defined limits.

Previous versions had implementation limits on

- How deep include files can `Include` another include file
- How deep `If/Ifdef/For` blocks can be nested
- How deep a macro can `Expand` another macro

These limits were arbitrary and were to guarantee that either Unimal cannot process your input or the results of the processing are independent of the machine resources available.

Practical experience has demonstrated that artificial limits are of little (or no) value: if Unimal fails due to lack of memory (on today's machines, a very unlikely event), you will know it. If you insist on using that machine, you'll need to modify the input file.

So, the previous implementation limits have been dropped.

6.2. Tokenization

When Unimal tokenizes the input file, it looks for the longest token it can find, just like C.

For instance, a statement

```
#MP remainder = dividend%dvisor
```

is not the same as

```
#MP remainder = dividend % divisor
```

In the former, the right-hand side is a composite name with the prefix `dividend` and the suffix `ivisor` rendered as a decimal number, according to the `%d` format.

It may look strange, but it is no different from C's

```
x = y /*p; /* something */;
```

where intended division by a number pointed to by `p` is in fact an opening of a comment.

As a general recommendation, simply do not save on blank spaces.

6.3. Including an output file

The following sequence will probably not work:

```
#MP Export (x) "foo"
Something That Goes To foo
#MP Include "foo" ;re-process the output
```

The reason is that by the time the include statement is reached Unimal didn't bother to close the file `foo`, on the grounds that it remains the current output stream. Depending on the habits of the operating system, the include statement may fail or it can include an empty or only partially written file.

To fix the problem, we have to switch the output away from `foo`, like

```
#MP Export (x) "foo"  
Something That Goes To foo  
#MP Export (0) "" ;to standard output  
#MP Include "foo" ;re-process the output
```

While this works, a better way to have a temporary output file is to push and pop the previous output stream:

```
#MP Export Push  
#MP Export (x) "foo"  
Something That Goes To foo  
#MP Export Pop ;closes foo and restores the previous output stream  
#MP Include "foo" ;re-process the output
```