Introduction to ANSYS Fluent scripting – Part 1 – Introduction to Scheme

If you have questions regarding this document, please try to contact the author, first:
akram.radwan@ansys.com

Description

The Scheme programming language is mentioned in the ANSYS Fluent documentation [1] [2] and also in some self-help knowledge materials available on the ANSYS Customer Portal. But what are Scheme scripts and what is the difference to journal files? How to access Fluent cell zone or boundary conditions within a script? How to create user-defined menu items, keyboard shortcuts or panels?

These questions are answered in this three-part series.

This first document describes Scheme. In principal you can find the content also in several digital books [3] [4] [5] [6] [7] but not all of the commands and procedures described there can be used in ANSYS Fluent. The goal of this first document is to give you a basic understanding of the Scheme programming language. It is not a complete reference guide but you can write most of your scripts with what you learn here.

The second document [8] describes Fluent specific commands and procedures. You learn how to process text output and some convenience procedures. That document can only scratch the surface but it should give you the tools that you need to develop versatile scripts for ANSYS Fluent. You should be familiar with the contents of the first document before you start with the second part.

With the third document [9] you can learn how to create your own menu items, keyboard shortcuts and panels. Most of this document is also covered in the official documentation but the examples might make it easier to understand the concept. You should be familiar with the contents of the first document before you start with the third part. Knowledge about the second part is not required.

Although everything has been tested with Fluent 17.0, 17.1 and a prototype of 18.0, it is possible that you get unexpected results. Scheme is not documented and Fluent procedures and commands can change the behavior or stop working completely with every new minor release.

Important

Scheme is poorly documented and not covered by ANSYS support contracts. ANSYS can refuse to provide support if you have trouble with simulations that rely on Scheme scripts. ANSYS can also refuse to provide support for journal files that contain Scheme code.

Nevertheless, with Scheme you have a mighty tool at your disposal. You can automate simulation setup, post-processing and case modifications. You can even create your own menu items and panels to have quicker access to panels and settings you need most often.

Introduction

What is Scheme?

Scheme is a scripting programming language. It was derived from the language LISP (LISt Processor) in 1975. A Scheme interpreter is the core of the ANSYS Fluent graphical user interface (GUI). Essentially, everything that can be done in the GUI can also be done with Scheme commands and procedures.

There is a rudimentary documentation available in the ANSYS Fluent Customization Manual, part II, chapter "Scheme Basics". This part of the documentation was introduced with Fluent 16. Apart from this very rudimentary and specific introduction there is no official documentation of Scheme for Fluent available.

With Scheme you can create scripts to automate different tasks. It starts with passing on variables to TUI (text user interface) commands. But it doesn't stop there. You can do complex automation tasks like solution-dependent adaption of simulation parameters with Scheme scripts.

Scheme is not the right language if you want to steer Fluent from a third-party application. This is the domain of Fluent as a Server (Fluent aaS) which is not covered in this document. Fluent aaS has its own manual, is well documented and you can get technical support.

What are journals?

A journal is a list of commands that Fluent executes from top to bottom. Journal files are simple text files with the file extension *.jou. The commands inside a journal can be simple text commands like `/solve/iterate 50`. But you can also use Scheme commands exclusively or in combination with text commands.

You need a journal if you want to run Fluent in batch mode. But they can also be useful if you want to reproduce certain tasks over and over again.

You can record a journal but in general ANSYS recommends to write your journals with text commands instead. Recorded journals contain Scheme commands that reproduce most of the operations you do in the Fluent GUI. For example, this is a recording where someone tried to calculate one time step with a time step size of 0.001 seconds and 20 iterations for each time step:

```
(cx-gui-do cx-set-list-tree-selections "NavigationPane*List_Tree1" (list "Solution|Run Calculation"))
(cx-gui-do cx-activate-item "Run Calculation*Table1*PushButton22(Calculate)")
(cx-gui-do cx-activate-item "Information*OK")
(cx-gui-do cx-activate-item "MenuBar*WriteSubMenu*Stop Journal")
```

Note: Line numbers and text colors are added to the examples that you can read and understand the examples better. They are not part of a journal or Scheme file. In this document Scheme commands, procedures and operands are blue while text commands are green. Strings are yellow unless they contain text or Scheme commands. Everything else is black.

When looking at the code above you might notice that it is difficult to read. Several commands seem to be nested and changing a specific value can be difficult because it is difficult to find it. Furthermore, you are probably missing the values mentioned above.

Besides readability and execution speed this is the largest downside of recorded journals. Although the user typed in all the values, Fluent did not record them because all fields were populated with the same values already. Running the journal with a different case might have a completely different result.

Now, the same attempt with text commands is just two lines:

```
/solve/set/time-step 0.001
/solve/dual-time-iterate 1 20
```

This is much cleaner and much easier to read by a human. And you can modify the settings easily for other simulations. It might take a bit longer to find the correct text commands compared to a recorded journal. But the improved robustness is well worth the effort.

However, there are some cases when recorded journals are very useful. This is covered in parts 2 and 3 od Introduction to ANSYS Fluent scripting.

If you are not familiar with the Fluent TUI (text user interface), you should read more about it in the ANSYS Fluent User's Guide, chapter "Text User Interface (TUI)". There are about 10 pages about how you can use the TUI and you should understand the concept before trying to modify or write Scheme scripts.

As you have seen above, Fluent journals can contain Scheme commands which makes it easy to mix in a little scripting into a journal to extend its capabilities.

What is the difference between journal files and Scheme files?

Essentially, both are text files with different file extensions. Journals use *.jou while Scheme scripts use *.scm. Both can contain Scheme code but only journal files are allowed to have Fluent text commands mixed in. You cannot use text commands within a Scheme loop or a user-defined Scheme procedure, though. For these cases you can encapsulate text commands with a Scheme command.

Fluent allows the execution of only one journal file. There is no restriction on Scheme scripts. They can even call themselves.

You can read both file types from the backstage view: File > Read > Journal… or File > Read > Scheme…

The text commands are `/file/read-journal` and `/file/read-macro`, respectively.

Scheme basics

In this chapter you learn how to use Scheme. There are also some official documentations available:

- Fluent Customization Manual, part II, chapter “Scheme basics” [2]
- Fluent User’s Guide, chapter “Scheme evaluation” [1]
- Fluent User’s Guide, chapter “Text prompt system” and its subchapters [1]

The description is very firm and does not go into much detail, though.

To test your basic Scheme code, you need access to ANSYS Fluent. You don’t need to read a case unless you need access to the Fluent data structure. You can type in all your code right in the TUI.

When you want to write more complex code, use a text editor that can keep track of parentheses (e.g. vi, emac or Notepad++). Scheme makes heavy use of parentheses. It’s easy to forget to close them at the right position especially when you’re optimizing your code.

Save your code with the file extension *.scm in your Fluent working directory and use either the backstage view or the text command to read it. If you need to read it often, you can create an alias to avoid going through the menu all the time. See part 2 of Introduction to ANSYS Fluent scripting for a description of how to create an alias for Scheme commands.

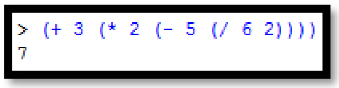
Calculating with Scheme

You can use the basic arithmetic operations by putting the operator first. `(+ 3 5)` calculates the sum of 3 + 5. Fluent prints the result to the console right after you hit return.

You can combine several operations by substituting one of the numbers with another operation:

```
(+ 3 (* 2 (- 5 (/ 6 2))))
```

This expression evaluates to 7:



```
> (+ 3 (* 2 (- 5 (/ 6 2))))
7
```

3 + (2 * (5 – (6 / 2))) =

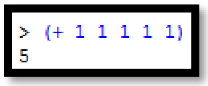
3 + (2 * (5 – 3)) =

3 + (2 * 2) =

3 + 4 = 7

As you can see, there is no priority of the operators like you might be used to from other languages. Everything is strictly evaluated from left to right while considering all parentheses.

You are not restricted to just two arguments. To build the sum over five number you can combine them in a single statement: `(+ 1 1 1 1 1)`



```
> (+ 1 1 1 1 1)
5
```

Of course there are more procedures for mathematical operations available. The following table shows the examples for the most important ones. The numbers can be replaced with variables and other statements.

Operation	Example	Fluent output

Negation	(- 5)	`> (- 5)`
		`-5`

Subtraction	(- 5 3 1)	`> (- 5 3 1)`
		`1`

Summation	(+ 1 2 3)	`> (+ 1 2 3)`
		`6`

Reciprocal	(/ 5)	`> (/ 5)`
		`0.2`

Division	(/ 6 3)	`> (/ 6 3)`
		`2`

Multiplication	(* 2 3)	`> (* 2 3)`
		`6`

Power	(expt 2 3)	`> (expt 2 3)`
		`8`

Maximum	(max 1 4 6 3)	`> (max 1 4 6 3)`
		`6`

Minimum	(min 1 4 6 3)	`> (min 1 4 6 3)`
		`1`

Absolute value	(abs -5)	`> (abs -5)`
		`5`

Square root	(sqrt 9)	`> (sqrt 9)`
		`3`

Natural logarithm	(log 2.81)	`> (log 2.81)`
		`1.0331845`

Exponential function	(exp 1)	`> (exp 1)`
		`2.7182818`

Sinus	(sin 1.5708)	`> (sin 1.5708)`
		`1`

Cosinus	(cos 0)	`> (cos 0)`
		`1`

Tangent	(tan 1)	`> (tan 1)`
		`1.5574077`

Arcsine	(asin 1)	`> (asin 1)`
		`1.5707963`

Arccosine	(acos 1)	`> (acos 1)`
		`0`

Arctangent	(atan 1)	`> (atan 1)`
		`0.78539816`

Remainder	(remainder 5 3)
	(modulo 5 3)
		`> (remainder 5 3)`
		`2`

Truncate	(truncate -4.7)	`> (truncate -4.7)`
		`-4`

Round	(round -4.7)	`> (round -4.7)`
		`-5`

Next smaller integer	(floor -4.3)	`> (floor -4.3)`
		`-5`

Next larger integer	(ceiling -4.3)	`> (floor -4.3)`
		`-5`

Note: There are even more operations that you can use. Check the Scheme books referenced at the end of this document for more. But keep in mind that not all of operations and procedures you find in public Scheme documentation work in Fluent.

White space and comments

Usually Fluent ignores unnecessary white spaces unless they are within a string. You can use as many white spaces as you need to improve the readability of your code.

```
(+ 3 (* 2 (- 5 (/ 6 2))))
```

You have seen this line in the last chapter. It is easy to get lost in the large number of parentheses required even for very simple scripts. Therefore, you can break up this single line into several lines to easily see the structure and how the different lines depend on each other.

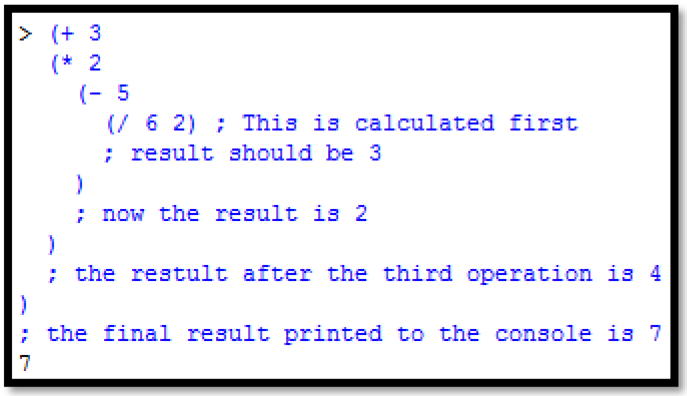
```
(+ 3
  (* 2
    (- 5
      (/ 6 2))))
```

It is good practice to put all closing parentheses right next to each other but that can be confusing if your script is several screens long. You can break them up if it helps you to structure your code and keep track of the parentheses.

```
(+ 3
  (* 2
    (- 5
      (/ 6 2)
    )
  )
)
```

You introduce a comment with a semicolon. The rest of the line will be ignored by Fluent.

```
(+ 3
  (* 2
    (- 5
      (/ 6 2) ; This is calculated first
      ; result should be 3
    )
    ; now the result is 2
  )
  ; the restult after the third operation is 4
)
; the final result printed to the console is 7
```



```
> (+ 3
  (* 2
    (- 5
      (/ 6 2) ; This is calculated first
      ; result should be 3
    )
    ; now the result is 2
  )
  ; the restult after the third operation is 4
)
; the final result printed to the console is 7
7
```

Introducing variables

There are three different types of variables. Scheme variables, RP variables and CX variables.

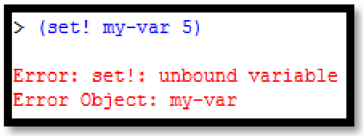
Scheme variables

Scheme variables are temporary variables. You can define them in your script as global or local variables. They reside only in memory. Global Scheme variables stay in memory as long as the Fluent process runs. They are not cleared when you read case or dat files but they are not saved with them. If you need a value to survive a Fluent restart, you can use RP variables.

There are three different procedures for defining, setting and displaying variables.

`(define my-var 3)` declares a new (global) variable and defines an initial value. It is imperative to specify an initial value, otherwise Fluent does nothing and the variable is not declared.

`(set! my-var 5)` sets the value of a global or local variable. The variable has to be declared already or Fluent reports an error about an unbound variable:



```
> (set! my-var 5)

Error: set!: unbound variable
Error Object: my-var
```

`(display my-var)` prints the value of a variable to the console or into a file (see chapter accessing files).

```
> (display my-var)
3
```

Local variables are covered later with the introduction of procedures and the `lambda` and `let` environments.

The name of a variable has to start with a letter. Afterwards you can use letters, numbers and even minus, slash or star characters. To avoid collision with variables and procedures that are pre-defined by Fluent, you can start all your variables with “my-” or something similar. Alternatively, you can check if a symbol is already in use (see end of chapter Procedures).

Scheme itself is case insensitive but it is recommended to use only lowercase letters in Scheme.

Scheme uses a dynamic type system. That means that the type of a variable can change dynamically, for example from a number to a string.

```
(define my-var 4)
(set! my-var "Hello")
```

The different data types are covered in chapter Data types.

RP variables

RP variables are usually solver parameters. RP stands for Rampant which was the name of one of predecessor solvers Fluent evolved from. RP vars are described in the Fluent Customization Manual, part I, chapter “Scheme Macros” and part II, chapter “RP Variables” [2].

They can be used to pass information from Scheme scripts to UDFs (User-Defined Functions) and vice versa. Fluent saves RP variables in the case file. They are persistent, though. If you define a RP var and load a new case, that RP var is still available. But if you close Fluent without saving the case, the RP var will be lost.

It is not easy to get rid of RP vars once they are saved in a case. You have to manually edit the case file with a text editor to eliminate them. Therefore, you should be careful when using RP vars and always use a new Fluent session before loading a new case where you do not need them anymore.

Important

Due to the connection with UDFs, RP vars have to be lowercase. Otherwise access from UDFs can fail.

The statement to define a RP var is a bit more complicated compared to Scheme variables:

```
(rp-var-define 'my-int 1 'integer #f)
```

To set the value of a RP var use:

```
(rpsetvar 'my-int 2)
```

To read the value of a RP var use:

```
(%rpgetvar 'my-int)
```

The `%` symbol forces Fluent to check for the stored value of the variable instead of the buffered value. Access to buffered values is faster and as long as you do not interact with UDFs you can neglect the `%` symbol. But when a UDF changes the value of a RP var the buffered value might not be outdated.

You can see from this example, that RP vars use a fixed type that is also available in the language C. This is required because RP vars are also accessibly by UDFs that are written in C. Name and C data type (integer, real, string) are passed as symbols, not as strings. This difference will become more clear after the introduction of Scheme data types. For more details, refer to the Fluent Customization Manual.

You can read more about RP vars in the documentation (see first paragraph of this chapter).

CX variables
CX variables have something to do with the GUI or the viewport. CX is short for cortex which is the process of the user interfaces (GUI, TUI, Scheme interpreter). Some of the CX variables are stored in a case file, some are just in memory. It is rare that you need access to CX variables. You cannot define your own CX variables.

To read CX vars use:

```
(cxgetvar 'cmap-list)
```

To set a CX var use:

```
(cxsetvar 'def-cmap "rgb")
```

Data types

Scheme uses a dynamic type system. The data type of a variable can change at any time. There are four basic and three compound data types.

The basic data types are:

- Boolean
- Number
- Character
- Symbol

Compound data types are:

- Strings
- Vectors
- Dotted pairs and lists

Booleans

Booleans take the value true or false, represented by `#t` and `#f`, respectively. You can also test if a value or a variable is of type Boolean:

```
(boolean? 3)                    → #f
```

You can also negate a Boolean value with the procedure `not`

```
(not #f)                        → #t
```

Numbers

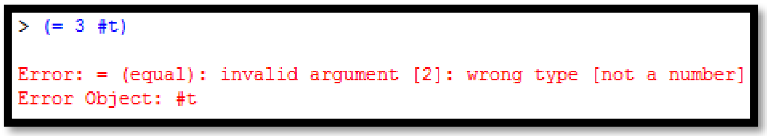
Numbers can be integer, floating point or complex numbers. Integers are always exact while floating point numbers are not exact. The mathematical procedures mentioned in chapter Calculating with Scheme work a little different if you feed them integer or floating point numbers. But usually you do not have to bother what type a number is.

You can check if a number is of a certain type. The following commands return #t when the type is correct:

```
(number? 3)                     → #t
(real? 3)                       → #t
(rational? 3)                   → #t
(integer? 3)                    → #t
```

To compare two numbers, you can use the equal sign or a procedure. The procedure is preferred because it can also compare to other data types while the equal sign can compare only numbers.

```
(eqv? 3 #t)                     → #f
(= 3 3)                         → #t
(= 3 #t)                        → error
```



```
> (= 3 #t)

Error: = (equal): invalid argument [2]: wrong type [not a number]
Error Object: #t
```

Other number comparisons that work only on numbers are: `<`, `<=`, `>`, `>=`. You will need them later when you use conditionals.

Note: Error messages can contain useful information. In this case the message tells you that the equal operator does not work because of an invalid argument. The problem is in the second argument, which is a Boolean value. The equal operator requires all arguments to be of the type number.

Characters

Characters are represented by a symbol combination. The letter “a” is written as `#\a`. `#\space`, `#\tab` and `#\newline` are self-explanatory. It is very rare that you need to work with single characters. You come across characters when you search for certain characters in a longer string. There are some commands that can be useful when working with characters:

```
(char? #\tab)                  → #t
(char=? #\a #\A)               → #f
(char-ci=? #\a #\A)            → #t
(char-downcase #\A)            → a
(char-upcase #\a)              → A
```

Characters by themselves are pretty useless. It is not even possible to display the value of a variable that contains a character. They become important in combination with strings. You can convert a character into a single-letter string with `make-string`. To get a character out of a string you can use `string-ref`. See also chapter Strings.

```
(make-string 1 #\a)            → a
(string-ref "Hello" 1)         → e
```

Symbols

Symbols are a sequence of characters that are used as identifier. Be careful not to confuse them with strings. For example, the name of a variable or a procedure is a symbol. For most other data types are self-evaluating. The number 3 is always the number 3. This is not the case for symbols. They usually evaluate to a value that is assigned to that symbol. The value can be one of the other data types but it can also be a complete procedure or a reference (e.g. to a port) that cannot be displayed easily. If you want to get the symbol, you need to quote it.

Example:

`(define my-var 3)` defines a variable with the name “`my-var`”. “`my-var`” is a symbol for the value 3, at least at the moment. When you use the characters “`my-var`” in your script it evaluates to its current value, in this example the number “3”. If you need access to the variable name itself, you use either `(quote my-var)` or `'my-var`.

Symbols are commonly used everywhere by Fluent. For example, if you access post-processing surfaces you need to use symbols. In that case the symbols (= name of the surfaces) hold a value that is a reference to something in the internal Fluent data structure. Usually you are not interested in that reference but in the name itself. Therefore, it is important to get familiar with symbols.

Of course, you can check if something is a symbol:

```
(symbol? 'my-var)              → #t
```

Fortunately, you do not need to bother about symbols in most cases. Many Fluent procedures acceps symbols and strings as input arguments. However, Fluent usually returns symbols and you might need to convert them to strings in order to work with them. Furthermore, symbols are required when working with RP and CX variables.

Strings

Strings are a sequence of characters. In fact, even a single character can be treated as a string. Do not confuse strings with symbols although they might look identical in the Fluent console.

The easy way to define strings is to put them in double quotes:

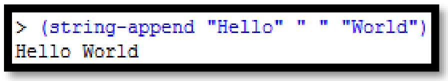
```
(define my-var "Hello")
```

Since strings are introduced using double quotes, you cannot use the double quotes inside a string without an escape character, which is the backspace character (marked red for your convenience):

```
(set! my-var "\"Hello\"")
```

To combine two or more strings, you can use

```
(string-append "Hello" " " "World")
```



```
> (string-append "Hello" " " "World")
Hello World
```

Of course you can also check if something is a string:

```
(string? 3)                      → #f
(string? "3")                    → #t
```

To compare two strings, you can use the procedures:

```
(fnmatch "Hel" "Hal")            → #f
(string=? "Hel" "hel")           → #f
(string-ci=? "Hel" "hel")        → #t
```

For strings, there is no difference of the result between the first two procedures. `fnmatch` also converts the datatype from symbol to string. `string=?` can only work with strings and therefore it's faster. `string-ci=?` ignores the case of the strings.

Often you need to convert a number to a string to build a text command that you can pass on to Fluent. It is not possible to combine strings and numbers without data type conversion. There are two methods available:

```
(string-append "Number " (number->string 5))
(format #f "Number ~a" 5)
```

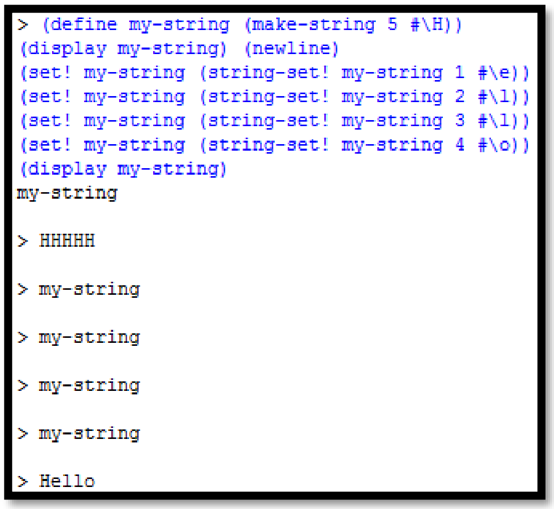
Both are covered in chapter Data type conversion.

You can create a string with a defined length with `make-string`.

```
(make-string 5 #\H)              → HHHHH
```

When you have the string, you can replace certain characters.

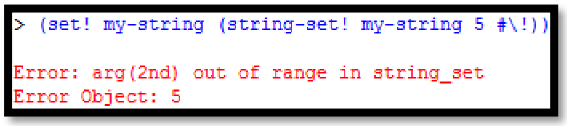
```
(define my-string (make-string 5 #\H))
(display my-string) (newline)
(set! my-string (string-set! my-string 1 #\e))
(set! my-string (string-set! my-string 2 #\l))
(set! my-string (string-set! my-string 3 #\l))
(set! my-string (string-set! my-string 4 #\o))
(display my-string)
```



```
> (define my-string (make-string 5 #\H))
(display my-string) (newline)
(set! my-string (string-set! my-string 1 #\e))
(set! my-string (string-set! my-string 2 #\l))
(set! my-string (string-set! my-string 3 #\l))
(set! my-string (string-set! my-string 4 #\o))
(display my-string)
my-string

> HHHHH

> my-string

> my-string

> my-string

> my-string

> Hello
```

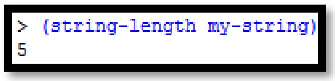
Line 1:	Define the variable "`my-string`" with an initial value of a string that consists of 5 times the character "H".
Line 2:	Show the value of "`my-string`" followed by a line break.
Line 3 - 6:	Replace all but the first character. Note that the first character starts at 0.
Line7:	Print the new value of "`my-string`" to the Fluent console.

You get an error if you try to replace a character that does not exist.



```
> (set! my-string (string-set! my-string 5 #\!))

Error: arg(2nd) out of range in string_set
Error Object: 5
```

You can also check the length of a string with `string-length` to avoid such error messages. Note that the length starts at 1 and not at zero like string-replace.

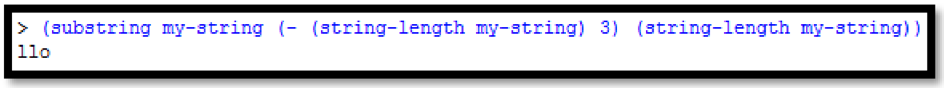


```
> (string-length my-string)
5
```

With `substring` you can extract a partial string from a given string. You need to specify the start and the end character. The start character is included; the end character is excluded.

```
(substring my-string 1 2)        → e
(substring my-string 1 3)        → el
```

Of course you can also calculate. For example, to get the last three characters you can combine substring and string length.



```
> (substring my-string (- (string-length my-string) 3) (string-length my-string))
llo
```

You can also convert a string from any case to lower case. Other conversions are not supported by Fluent.

```
(string-downcase my-string)→ hello
```

If you want to check if certain characters exist in a string, you have two options. `string-ref` gives you access to individual characters. `string->list` converts a string into a list of characters that you can process with all the tools available for lists.

`(string-ref my-string 1)` → e

Note: string-ref always returns a single character. substring always returns a string. That string can consist of a single character. But strings and characters are different data types and procedures that work with strings cannot process characters and vice versa.

Vectors

Vectors come in handy if you need vector operations. The definition of a vector uses this keyword:

`(vector 1 2 3)`

The notation for the result is: `#(1 2 3)`

The dimension of a vector is not limited.

It is not necessary that each element of a vector is a number. You can mix data types in a vector just like you can do it with lists. Although vectors require less space and are faster to access, Fluent usually uses lists instead of vectors.

There are several procedures available to work with vectors. Most of them are self-explanatory.

`(vector? '#(1 2 3))` → #t

This is identical to `(vector? (vector 1 2 3))` but uses a symbol notation instead of passing a value.

You can also report the length of a vector.

`(vector-length '#(1 2 3))` → 3

Note that only the primary dimension is given. If each element of a vector is another vector, this cannot be shown directly.

You can also access a certain element of a vector. The first element is treated as element number zero.

`(vector-ref '#(1 2 3) 1)` → 2

Of course you can also set an element

`(vector-set! '#(1 2 3) 1 5)` → #(1 5 3)

Dotted pairs and lists

Dotted pairs and lists are a very important concept. Fluent stores a lot of information in lists of symbols. Lists are similar to vectors but there are different procedures available to work with them.

The base structure of a list is a dotted pair. To create a pair, you need the cons-statement

`(define my-p (cons 1 "a"))`

This creates a dotted pair of the number 1 and the string “a”.

The first element of the pair is called `car`, the second `cdr`:

`(car my-p)` → 1

`(cdr my-p)` → a

If the second element is also a dotted pair you can create a list. Instead of nesting many `cons` statements together you can use the list statement:

`(define my-l (list 1 "a" 3 4))` or `(define my-l '(1 "a" 3 4))`

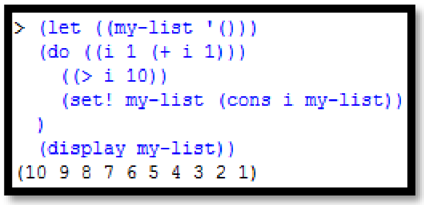
`(car my-l)`	→ 1
`(cdr my-l)`	→ `(a 3 4)`
`(car (cdr my-l))`	→ a

The last one can be abbreviated with `cadr`. Up to four combinations of `a` and `d` are possible.

`(cadr my-l)`	→ a
`(cadddr my-l)`	→ 4

Nested dotted pairs are only a list if the last element is an empty list. This is important when you create your own lists with loops or recursions.
The let environment and the do loop used in this example is discussed later.

```
(let ((my-list '()))
  (do ((i 1 (+ i 1)))
    ((> i 10))
    (set! my-list (cons i my-list))
  )
  (display my-list))
```



```
> (let ((my-list '()))
  (do ((i 1 (+ i 1)))
    ((> i 10))
    (set! my-list (cons i my-list))
  )
  (display my-list))
(10 9 8 7 6 5 4 3 2 1)
```

Lists are important because Fluent can return many items as list. For example,
`(inquire-surface-names)`
returns a list of all surfaces that can be used for post-processing. Note that Fluent returns a list of symbols, not a list of strings. The difference is not visible from the output.

You can loop through all elements of a list with a `for-each` loop. This will be covered in chapter For-each loop.

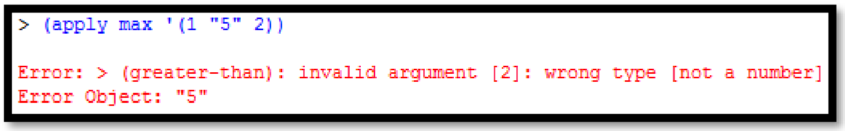
Of course you can also check if something is a list or a pair. Fluent does not distinguish between both types for this check.

`(pair? my-p)`	→ #t
`(pair? my-l)`	→ #t

To calculate with complete lists you can use the procedure apply.

`(apply max '(1 5 2))`	→ 5

Of course, all items of the list must be numbers. If you have a list of mixed types, you get an error.



```
> (apply max '(1 "5" 2))

Error: > (greater-than): invalid argument [2]: wrong type [not a number]
Error Object: "5"
```

There are some additional useful procedures available when you work with lists.

`(length '("a" 1))`	→ 2
`(list-ref '("first" "second" "third" "fourth") 2)`	→ third
`(list-tail '(1 2 3 4 5 6) 2)`	→ (3 4 5 6)
`(append '(1 2 3) '(4 5))`	→ (1 2 3 4 5)
`(reverse '(3 2 1))`	→ (1 2 3)
`(member 3 '(1 2 3 4 5 6))`	→ (3 4 5 6)
`(member 3 '(1 2 4 5 6))`	→ #f

`length` returns the number of elements of a list.

`list-ref` returns the specified element of a list. It starts to count at 0.

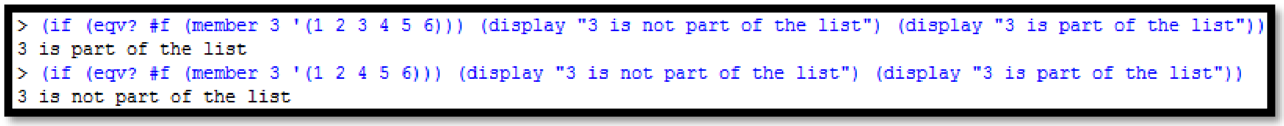
`list-tail` returns a list with all elements starting at the specified element. It starts to count at 0. `(list-tail '(…) 1)` is identical to `(cdr '(…))`.

`append` combines two lists into one.

`reverse` reverses the order of a list.

`member` returns the tail starting with the specified object. It returns `#f` if the object cannot be found. You can also use it to check if a single item exists in the list.

```
(if (eqv? #f (member 3 '(1 2 3 4 5 6)))
  (display "3 is not part of the list")
  (display "3 is part of the list"))
```



```
> (if (eqv? #f (member 3 '(1 2 3 4 5 6))) (display "3 is not part of the list") (display "3 is part of the list"))
3 is part of the list
> (if (eqv? #f (member 3 '(1 2 4 5 6))) (display "3 is not part of the list") (display "3 is part of the list"))
3 is not part of the list
```

Data type conversion

You can convert most data types into each other.

`(string->number "5")` makes the string "5" into the number `5` that you can use in calculations.

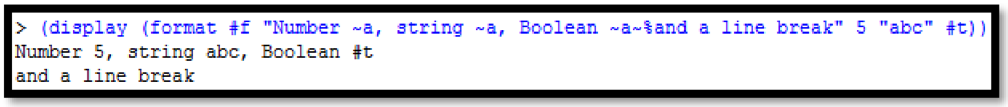
`(number->string 5)` does the opposite that you can append the number 5 to a string to build a text command to pass on to Fluent.

Another conversion that you might need from time to time is `(symbol->string 'my-var)` or `(string->symbol "my-var")`.

Another popular conversion is `list->vector` and `vector->list`.

The most important conversion when writing scripts for Fluent is the `format` procedure. Essentially this procedure is used for formatting, the conversion to a string is a by-product. The procedure can take a large number of arguments.

```
(display (format #f "Number ~a, string ~a, Boolean ~a~%and a line break" 5
"abc" #t))
```

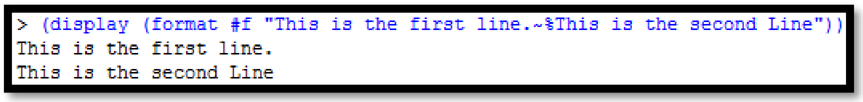


```
> (display (format #f "Number ~a, string ~a, Boolean ~a~%and a line break" 5 "abc" #t))
Number 5, string abc, Boolean #t
and a line break
```

The first parameter is the destination. For Fluent this is usually `#f`. There are cases when that Boolean value is not required but inside procedures it can provoke errors if it's not there.

The second parameter is a string with some special symbols. ~a fills in the arguments that follow the string in order. The number of ~a and the number of arguments must match or Fluent will report an error. You can also add a line break with ~%.

```
(display (format #f "This is the first line.~%This is the second Line."))
```



```
> (display (format #f "This is the first line.~%This is the second Line"))
This is the first line.
This is the second Line
```

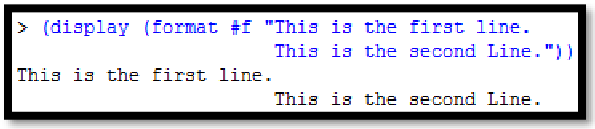
Alternatively, you can use `\n` to add a line break:

```
(display (format #f "This is the first line.\nThis is the second Line."))
```

Note: Line breaks with `\n` are ignored when you send such a string to a file.

You can also add the line break in the string directly. But you cannot add additional white spaces to improve the readability of your script. The ~ symbol that is referenced in some Scheme books to ignore white spaces within a string is not recognized by Fluent.

```
(display (format #f "This is the first line.
                     This is the second Line."))
```

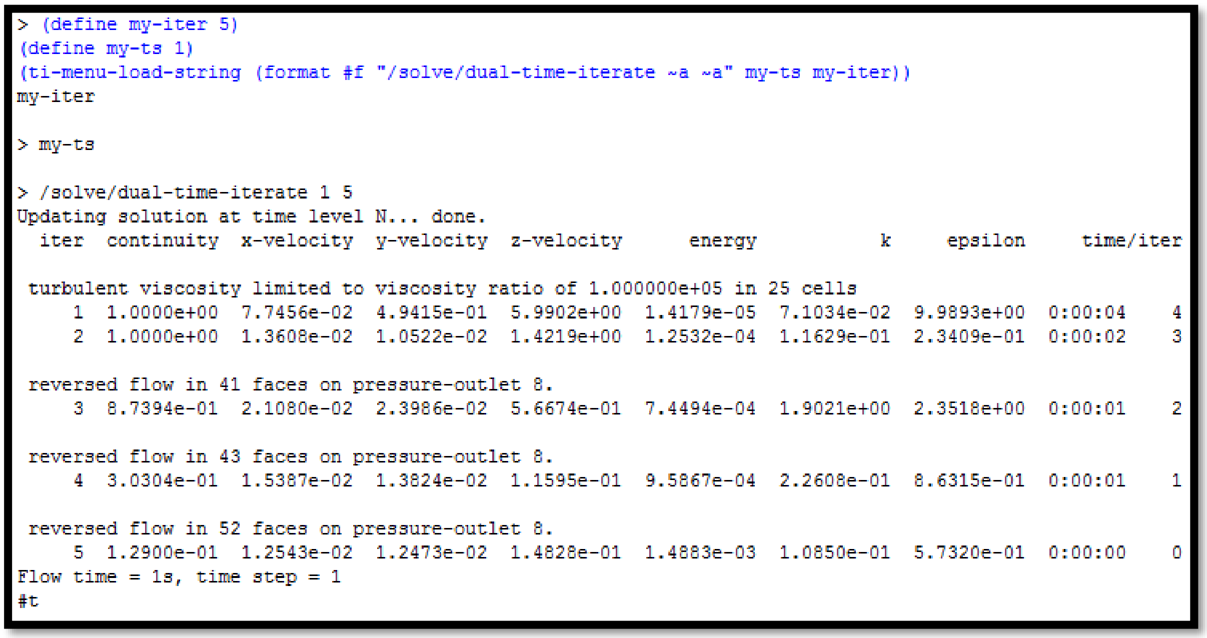


```
> (display (format #f "This is the first line.
                       This is the second Line."))
This is the first line.
                       This is the second Line.
```

The format procedure is often used when passing the value of variables into a Fluent text command.

```
(define my-iter 5)
(define my-ts 1)
(ti-menu-load-string (format #f "/solve/dual-time-iterate ~a ~a" my-ts
my-iter))
```

These three lines define two variables and send the text command `/solve/dual-time-iterate` with two arguments to Fluent, telling it to run one time step with 5 iterations. Of course this works only when case and data are loaded already.

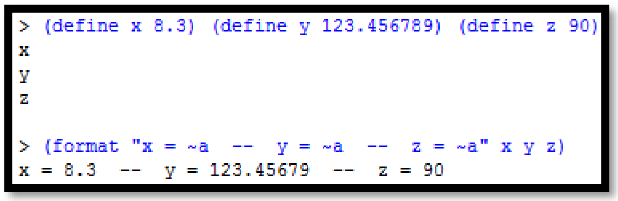


```
> (define my-iter 5)
(define my-ts 1)
(ti-menu-load-string (format #f "/solve/dual-time-iterate ~a ~a" my-ts my-iter))
my-iter

> my-ts

> /solve/dual-time-iterate 1 5
Updating solution at time level N... done.
  iter  continuity  x-velocity  y-velocity  z-velocity      energy           k     epsilon     time/iter

 turbulent viscosity limited to viscosity ratio of 1.000000e+05 in 25 cells
     1  1.0000e+00  7.7456e-02  4.9415e-01  5.9902e+00  1.4179e-05  7.1034e-02  9.9893e+00  0:00:04    4
     2  1.0000e+00  1.3608e-02  1.0522e-02  1.4219e+00  1.2532e-04  1.1629e-01  2.3409e-01  0:00:02    3

 reversed flow in 41 faces on pressure-outlet 8.
     3  8.7394e-01  2.1080e-02  2.3986e-02  5.6674e-01  7.4494e-04  1.9021e+00  2.3518e+00  0:00:01    2

 reversed flow in 43 faces on pressure-outlet 8.
     4  3.0304e-01  1.5387e-02  1.3824e-02  1.1595e-01  9.5867e-04  2.2608e-01  8.6315e-01  0:00:01    1

 reversed flow in 52 faces on pressure-outlet 8.
     5  1.2900e-01  1.2543e-02  1.2473e-02  1.4828e-01  1.4883e-03  1.0850e-01  5.7320e-01  0:00:00    0
Flow time = 1s, time step = 1
#t
```

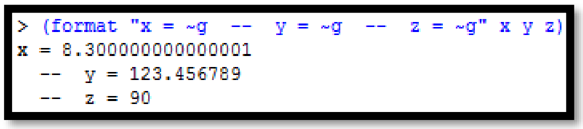
You can use format also to display formatted numbers.

~a	equivalent to display
~g	enhanced output including line break
~d	decimal number
~e	exponential notation
~f	floating point notation

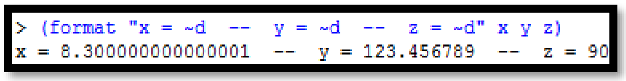
Examples:



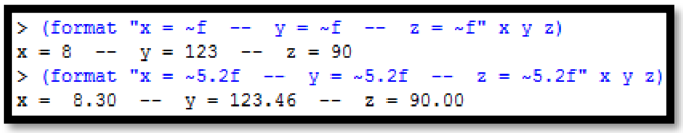
```
> (define x 8.3)  (define y 123.456789)  (define z 90)
x
y
z

> (format "x = ~a  --  y = ~a  --  z = ~a" x y z)
x = 8.3  --  y = 123.45679  --  z = 90
```



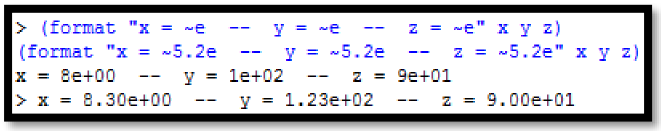
```
> (format "x = ~g  --  y = ~g  --  z = ~g" x y z)
x = 8.300000000000001
  --  y = 123.456789
  --  z = 90
```



```
> (format "x = ~d  --  y = ~d  --  z = ~d" x y z)
x = 8.300000000000001  --  y = 123.456789  --  z = 90
```



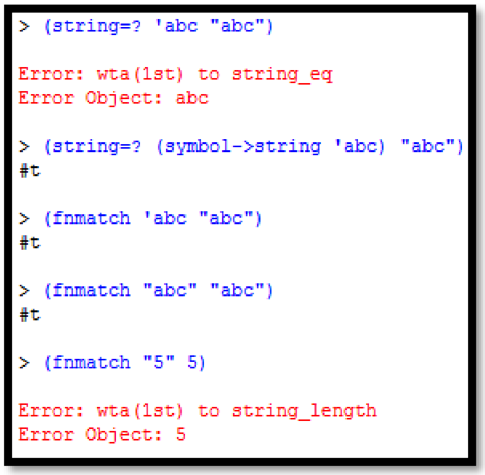
```
> (format "x = ~f  --  y = ~f  --  z = ~f" x y z)
x = 8  --  y = 123  --  z = 90
> (format "x = ~5.2f  --  y = ~5.2f  --  z = ~5.2f" x y z)
x =  8.30  --  y = 123.46  --  z = 90.00
```



```
> (format "x = ~e  --  y = ~e  --  z = ~e" x y z)
(format "x = ~5.2e  --  y = ~5.2e  --  z = ~5.2e" x y z)
x = 8e+00  --  y = 1e+02  --  z = 9e+01
> x = 8.30e+00  --  y = 1.23e+02  --  z = 9.00e+01
```

Note: You can specify the number of digits for `~f` and `~e`. `~5.2f` shows a total of 5 digits of which 2 are after the decimal point. However, if the integer part is larger than 3 digits, it will be extended. `~5.2e` shows two digits after the decimal point. The first number is disregarded. Fluent ignores the numbers if you use them with `~d`.

In Fluent it can be necessary to compare members of a list with a certain string. Many of the lists contain symbols instead of strings. You can use `fnmatch` to compare symbols to strings without doing the data type conversion yourself. It cannot compare a number to a string, though.



```
> (string=? 'abc "abc")

Error: wta(1st) to string_eq
Error Object: abc

> (string=? (symbol->string 'abc) "abc")
#t

> (fnmatch 'abc "abc")
#t

> (fnmatch "abc" "abc")
#t

> (fnmatch "5" 5)

Error: wta(1st) to string_length
Error Object: 5
```

Local variables

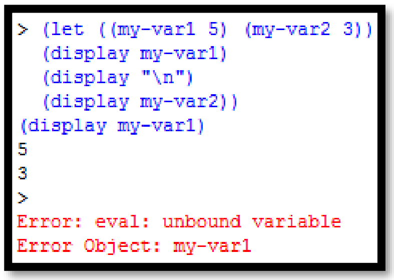
If you do not want your variable declaration to be global, you can use the `let` statement to limit their availability.

```
(let ((my-var1 5) (my-var2 3))
  (display my-var1)
  (display "\n")
  (display my-var2))
(display my-var1)
```

The first parentheses after `let` contains a list of variables that is only available until the let statement is closed. Because of this list you still need double parentheses even when you define only a single variable.

Note: There is no `define` statement. Just specify the name and the initial value. The number of elements in the list is not restricted.

The two variables defined in this example are not available after the closing parentheses of the let statement. Therefore, the last line provokes an error.

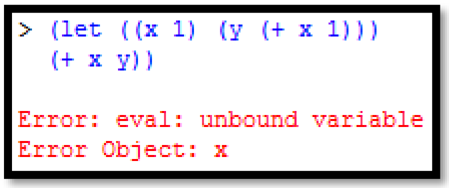

```
> (let ((my-var1 5) (my-var2 3))
  (display my-var1)
  (display "\n")
  (display my-var2))
(display my-var1)
5
3
>
Error: eval: unbound variable
Error Object: my-var1
```

Unless you need global Scheme variables, you should consider using local variables whenever possible. This avoids conflicts when you use different Scheme scripts with different case files in a single Fluent session.

There is a similar way of defining local variables when using your own procedures.

You cannot define dependent variables within a single `let` statement.

```
(let ((x 1) (y (+ x 1)))
  (+ x y))
```


```
> (let ((x 1) (y (+ x 1)))
  (+ x y))

Error: eval: unbound variable
Error Object: x
```

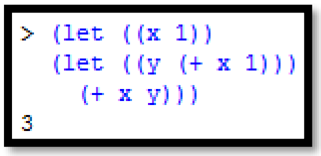
There are three ways to avoid the error:

1. Nest multiple `let` statements
2. Use `set!` to redefine a variable later
3. Use the statement `let*`

Nesting

You can combine multiple let statements after each other. Take care of all the parentheses because it's very easy to get lost.

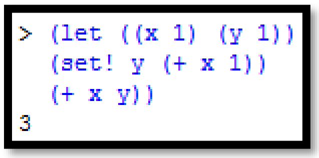
```
(let ((x 1))
  (let ((y (+ x 1)))
    (+ x y)))
```


```
> (let ((x 1))
  (let ((y (+ x 1)))
    (+ x y)))
3
```

Set!

You can also define all local variables in a single let statement and redefine the dependent variable later.

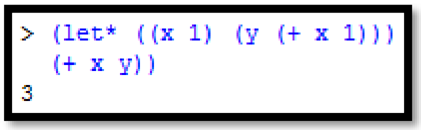
```
(let ((x 1) (y 1))
  (set! y (+ x 1))
  (+ x y))
```



```
> (let ((x 1) (y 1))
  (set! y (+ x 1))
  (+ x y))
3
```

Let*
Fluent allows dependent variables within a single let* statement. You save some space compared to the first two approaches.

```
(let* ((x 1) (y (+ x 1)))
  (+ x y))
```



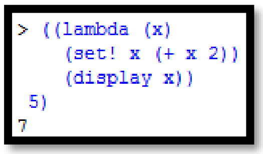
```
> (let* ((x 1) (y (+ x 1)))
  (+ x y))
3
```

Named let
You can give the let environment a name that it can call itself. This is covered in chapter recursion. You cannot call a named let from outside, though. For that you need procedures.

Procedures

You can think of a procedure as a list of one or several commands. A procedure does not necessarily have a name (= unique symbol). You define a procedure with a `lambda` environment. It is very similar to the let statement you have seen before.

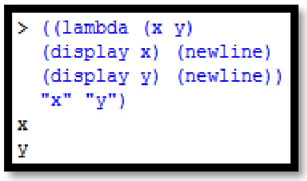
```
((lambda (x)
  (set! x (+ x 2))
  (display x))
  5)
```



```
> ((lambda (x)
    (set! x (+ x 2))
    (display x))
 5)
7
```

Fluent executes a procedure like that immediately. It looks very similar to let but there are some important differences. You specify one or multiple parameters in the parentheses behind lambda but you cannot define initial values. These initial values are passed as arguments at the end of the procedure.

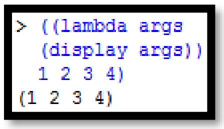
```
((lambda (x y)
  (display x) (newline)
  (display y) (newline))
  "x" "y")
```



```
> ((lambda (x y)
  (display x) (newline)
  (display y) (newline))
  "x" "y")
x
y
```

You cannot overload[1] a procedure but you can specify a variable number of arguments with the list concept.

```
((lambda args
  (display args))
  1 2 3 4)
```



```
> ((lambda args
  (display args))
  1 2 3 4)
(1 2 3 4)
```

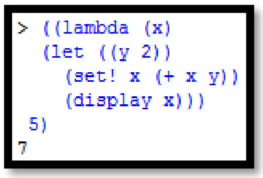
The arguments are turned into a list that you can work with.

Note: There are no parentheses around the variable name “args”. This means that all arguments are optional. If some arguments are mandatory you need to turn the last parameter into a dotted pair:

```
((lambda (x1 x2 x3 . args) (display args)) 1 2 3 "additional arguments
optional")
```

Of course, you can also combine lambda and let statements.

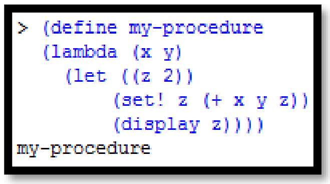
```
((lambda (x)
  (let ((y 2))
    (set! x (+ x y))
    (display x)))
 5)
```



```
> ((lambda (x)
  (let ((y 2))
    (set! x (+ x y))
    (display x)))
 5)
7
```

Procedures become powerful when you give them a name with `define`. Then you can use them over and over again and pass arguments instead of hard code them.

```
(define my-procedure
  (lambda (x y)
    (let ((z 2))
      (set! z (+ x y z))
      (display z))))
```



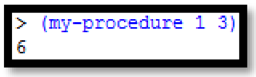
```
> (define my-procedure
  (lambda (x y)
    (let ((z 2))
      (set! z (+ x y z))
      (display z))))
my-procedure
```

[1] Overloading means that a procedure is defined multiple times, each time with a different number of arguments.

Note: This works just like with variables. But the value of symbol is not a single literal. Instead it is the code of the procedure.

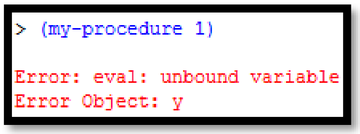
Fluent evaluates the code of the procedure when you call it.

```
001 (my-procedure 1 3)
```

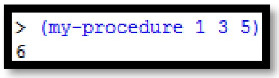


```
> (my-procedure 1 3)
6
```

You need to specify all required parameters or you get an error. Fluent ignores parameters that you specify on top.

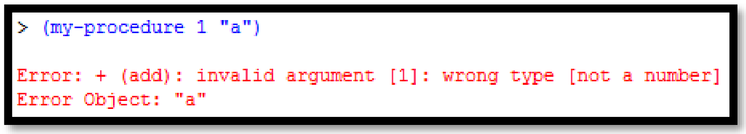


```
> (my-procedure 1)

Error: eval: unbound variable
Error Object: y
```



```
> (my-procedure 1 3 5)
6
```

You can run this procedure as often as you like with any allowed parameter. Of course, if you passed on and invalid data type you get an error message.

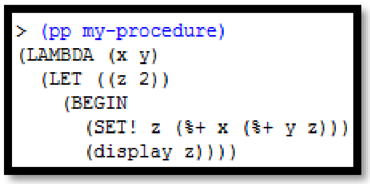


```
> (my-procedure 1 "a")

Error: + (add): invalid argument [1]: wrong type [not a number]
Error Object: "a"
```

Important

There is no difference on how Fluent works with named procedures and variables. You can assign any value to the symbol using the set! statement. Be careful that you do not overwrite a procedure with a literal by accident. Consistent naming conventions can help you to avoid such issues.

To print the code of a procedure, use the procedure pp. This is not restricted to user-defined procedures, but can also be used for procedures hardcoded into Fluent. You only need to know the name.

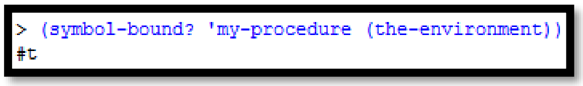


```
> (pp my-procedure)
(LAMBDA (x y)
  (LET ((z 2))
    (BEGIN
      (SET! z (%+ x (%+ y z)))
      (display z))))
```

Procedures stay in memory as long as the fluent session runs. They are not stored with the case file. If you need the same procedure over and over again, you can think about adding it to the .fluent file which is located in your home directory.

Just like the variable definitions you can create conflict by redefining a procedure that is required by Fluent. There is a procedure available to check if a symbol is already defined.

```
(symbol-bound? 'my-procedure (the-environment))
```



```
> (symbol-bound? 'my-procedure (the-environment))
#t
```

You can use this for procedures and variables alike. The parameter the environment is imperative. It contains all the symbol definitions in the active Fluent session.

Important

Never call `the-environment` by itself. In the best case you crash Fluent when calling it. In the worst case you could damage your system!

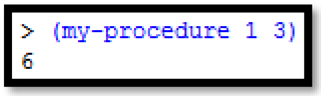
Important

Check your desired procedure names against the-environment before defining them. You can overwrite procedures that Fluent requires to run correctly. You can break Fluent completely if you are not careful about your procedure names.

Abbreviation

Since named procedures are often required Fluent knows a short form of the definition. You can put the name of the procedure together with the name for all parameters in parentheses after the `define` statement. Then it is no longer necessary to use the lambda environment.

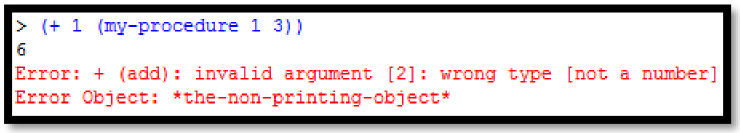
```
001  (define (my-procedure x y)
002    (let ((z 2))
003      (set! z (+ x y z))
004      (display z)))
```



```
> (my-procedure 1 3)
6
```

Return values

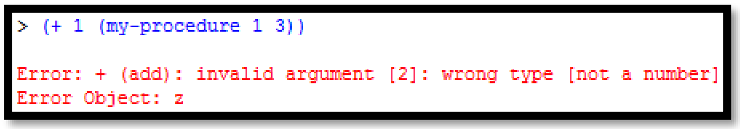
The last statement of a procedure is the return value. This is important if you need the result of one procedure for something else. However, not all statements give usable return values. The result of the procedure defined before cannot be used any further.



```
> (+ 1 (my-procedure 1 3))
6
Error: + (add): invalid argument [2]: wrong type [not a number]
Error Object: *the-non-printing-object*
```

The same is true if you remove line 4.

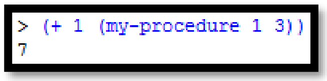
```
001  (define (my-procedure x y)
002    (let ((z 2))
003      (set! z (+ x y z))))
```



```
> (+ 1 (my-procedure 1 3))

Error: + (add): invalid argument [2]: wrong type [not a number]
Error Object: z
```

But it works if you remove the set! statement.

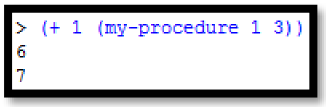
```
(define (my-procedure x y)
  (let ((z 2))
    (+ x y z)))
```



```
> (+ 1 (my-procedure 1 3))
7
```

If you want to keep the original output, you can just add the return value as last statement. The newline is required that there is some distance between the output of the procedure itself and the result of the next operation.

```
(define (my-procedure x y)
  (let ((z 2))
    (set! z (+ x y z))
    (display z) (newline)
    z))
```



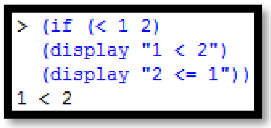
```
> (+ 1 (my-procedure 1 3))
6
7
```

Conditionals

If

The most common conditional is the `if` statement. It consists of a condition, some code that is executed if the condition becomes true, and optionally additional code that is executed when the condition is false.

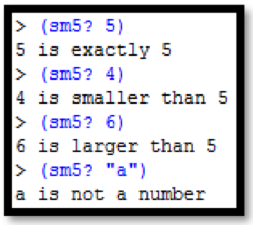
```
(if (< 1 2)
  (display "1 < 2")
  (display "2 <= 1"))
```



```
> (if (< 1 2)
  (display "1 < 2")
  (display "2 <= 1"))
1 < 2
```

If you need to execute more code than just a single line, you need the `begin` statement to encapsulate the code.

```
(define sm5?
  (lambda (x)
    (if (number? x)
      (begin
        (if (< x 5)
          (display (format #f "~a is smaller than 5" x))
          (begin
            (if (eqv? x 5)
              (display (format #f "~a is exactly 5" x))
              (display (format #f "~a is larger than 5" x))))))
      (display (format #f "~a is not a number" x)))))
```

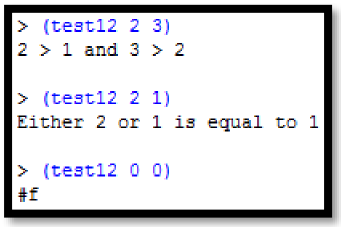


```
> (sm5? 5)
5 is exactly 5
> (sm5? 4)
4 is smaller than 5
> (sm5? 6)
6 is larger than 5
> (sm5? "a")
a is not a number
```

Line 1:	Definition of the name “`sm5?`” for the user-defined procedure.
Line 2:	Definition of a single parameter that can be passed as argument to the procedure.
Line 3:	Check if the parameter is a number.
Line 4:	The code block between line 4 and line 10 is executed if the condition in line 3 evaluates to `#t`.
Line 5:	Another `if` statement checks if the parameter is smaller than 5.
Line 6:	Text is printed to the Fluent console that the parameter is smaller than 5. Since this is a single line, no begin statement is required. This is executed if the condition in line 5 evaluates to `#t`.
Line 7:	The code block between line 7 and 10 is executed if the condition in line 5 evaluates to `#f`.
Line 8:	The last `if` statement checks if the parameter is equal to 5
Line 9:	Text is printed to the Fluent console that the parameter is equal to 5. This is executed if the condition in line 8 evaluates to `#t`.
Line 10:	Text is printed to the Fluent console that the parameter is larger than 5. This is executed if the condition in line 8 evaluates to `#f`.
Line 11:	Text is printed to the Fluent console that the parameter is not a number. This is executed if the condition in line 3 evaluates to `#f`. It is a single line of code, so no `begin` statement is required.

You can also use `and` and `or` to combine multiple checks in one statement. Of course you can use this outside of `if` statements, too.

```
(define (test12 a b)
  (if (and (> a 1) (> b 2))
    (display (format #f "~a > 1 and ~a > 2\n" a b))
    (if (or (eqv? a 1) (eqv? b 1))
      (display (format #f "Either ~a or ~a is equal to 1\n" a b)))))
```

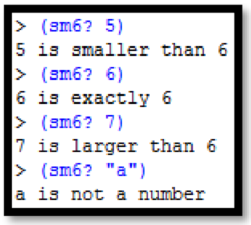


```
> (test12 2 3)
2 > 1 and 3 > 2

> (test12 2 1)
Either 2 or 1 is equal to 1

> (test12 0 0)
#f
```

Cond

The `cond` statement is handy when you have several conditions in a row. Like in the example above where several statements were nested.

```
001  (define sm6?
002    (lambda (x)
003      (cond ((number? x)
004        (cond
005          ((< x 6) (display (format #f "~a is smaller than 6" x)))
006          ((eqv? x 6) (display (format #f "~a is exactly 6" x)))
007          (else (display (format #f "~a is larger than 6" x)))))
008        (else (display (format #f "~a is not a number" x))))))
```



```
> (sm6? 5)
5 is smaller than 6
> (sm6? 6)
6 is exactly 6
> (sm6? 7)
7 is larger than 6
> (sm6? "a")
a is not a number
```

Line 1: Definition of the name “`sm5?`” for the user-defined procedure.

Line 2: Definition of a single parameter that has to be passed as argument to the procedure.

Line 3: Check if the parameter is a number. Line 4 is executed if it returns `#t`, line 8 for all other values.

Line 4: Start multiple comparisons.

Line 5: Check and print text to the Fluent console that the parameter is smaller than 6.

Line 6: Check and print text to the Fluent console that the parameter is equal 6.

Line 7: For all other cases print text to the Fluent console that the parameter is larger than 6.

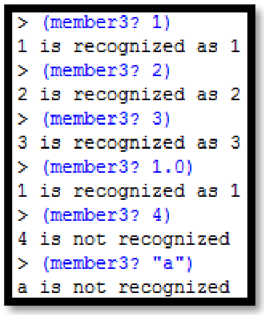
Line 8: For all other cases of the condition in line 3 print text to the Fluent console that the parameter is not a number.

When you use the `cond` statement you should take special care of the parentheses. Especially when you nest several `cond` statements it is very easy to lose track of all the closing parentheses.

Case

If you need to compare a parameter against specific values, you can use the `case` statement. It is a simplified version of the `cond` statement.

```
001  (define member3?
002    (lambda (x)
003      (case x
004        ((1) (display (format #f "~a is recognized as 1" x)))
005        ((2) (display (format #f "~a is recognized as 2" x)))
006        ((3) (display (format #f "~a is recognized as 3" x)))
007        (else (display (format #f "~a is not recognized" x))))))
```



```
> (member3? 1)
1 is recognized as 1
> (member3? 2)
2 is recognized as 2
> (member3? 3)
3 is recognized as 3
> (member3? 1.0)
1 is recognized as 1
> (member3? 4)
4 is not recognized
> (member3? "a")
a is not recognized
```

Line 1:	Definition of the name “`member3?`” for the user-defined procedure.
Line 2:	Definition of a single parameter that has to be passed as argument to the procedure.
Line 3:	Start the case check by providing the parameter.
Line 4:	Check parameter against the number 1 and print text if the check returns `#t`.
Line 5:	Check parameter against the number 2 and print text if the check returns `#t`.
Line 6:	Check parameter against the number 3 and print text if the check returns `#t`.
Line 7:	Print text if none of the earlier tests in lines 4 to 6 returned `#t`.

If you want to execute multiple lines of code for one of the conditions can use the `begin` statement like you have seen it for the `if` statement.

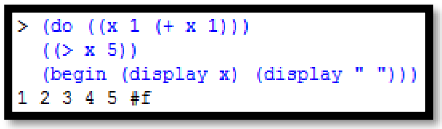
Iteration

There are two ways for iterations: looping and recursion.

Do loop

The simple form of the `do` loop takes one iteration variable and one abort condition. You do not have to declare the iteration variable before, it is defined as local variable inside the `do` loop automatically.

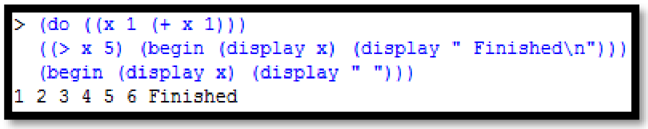
```
(do ((x 1 (+ x 1)))
  ((> x 5))
  (begin (display x) (display " ")))
```



```
> (do ((x 1 (+ x 1)))
  ((> x 5))
  (begin (display x) (display " ")))
1 2 3 4 5 #f
```

You can also add commands to execute when the abort condition becomes true.

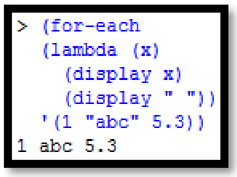
```
(do ((x 1 (+ x 1)))
  ((> x 5) (begin (display x) (display " Finished\n")))
  (begin (display x) (display " ")))
```



```
> (do ((x 1 (+ x 1)))
  ((> x 5) (begin (display x) (display " Finished\n")))
  (begin (display x) (display " ")))
1 2 3 4 5 6 Finished
```

For-each loop
Another common loop is the `for-each` loop. This loops over all items of a list.

```
001 (for-each
002   (lambda (x)
003     (display x)
004     (display " "))
005   '(1 "abc" 5.3))
```



```
> (for-each
  (lambda (x)
    (display x)
    (display " "))
  '(1 "abc" 5.3))
1 abc 5.3
```

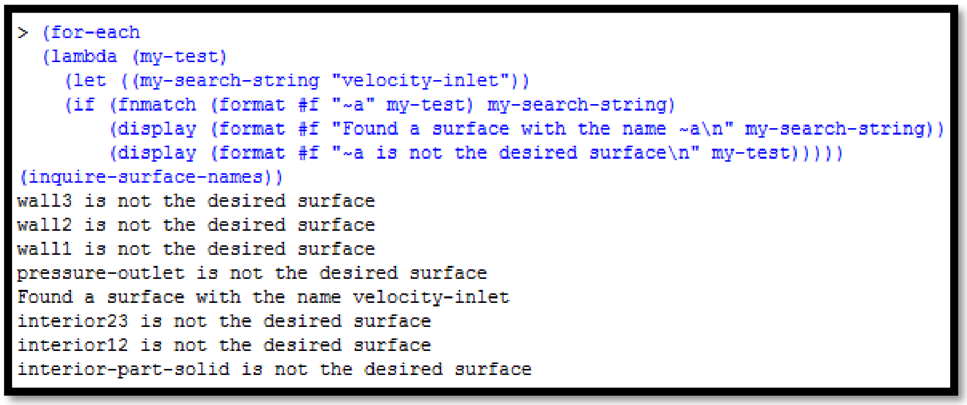
Line 1: Start the `for-each` loop.
Line 2: Definition of the environment. The variable “x” holds the current item of the list.
Line 3/4: Display the current item and a white space.
Line 5: Specify the list

This is how the loop works for this example:

1. Take the first item of the list (provided in line 5, here: `1`)
2. Store this item in the variable `x`. Only a single value is stored in `x`, not the whole list.
3. Display `x`, for the first loop this is `1`.
4. Display a white space to visually separate the display of the next loop
5. Continue with step 1, but use the second item of the list, the string `"abc"`…

Of course, instead of giving the list explicitly, you can also pass in a list as variable or procedure. This is commonly used to loop through lists that are provided by Fluent. In the following example a Fluent Scheme procedure is used that returns a list of all post-processing surfaces that exist in the case.

```
(for-each
  (lambda (my-test)
    (let ((my-search-string "velocity-inlet"))
      (if (fnmatch (format #f "~a" my-test) my-search-string)
        (display (format #f "Found a surface with the name ~a\n" my-search-string))
        (display (format #f "~a is not the desired surface\n" my-test)))))
  (inquire-surface-names))
```



```
> (for-each
  (lambda (my-test)
    (let ((my-search-string "velocity-inlet"))
    (if (fnmatch (format #f "~a" my-test) my-search-string)
        (display (format #f "Found a surface with the name ~a\n" my-search-string))
        (display (format #f "~a is not the desired surface\n" my-test)))))
(inquire-surface-names))
wall3 is not the desired surface
wall2 is not the desired surface
wall1 is not the desired surface
pressure-outlet is not the desired surface
Found a surface with the name velocity-inlet
interior23 is not the desired surface
interior12 is not the desired surface
interior-part-solid is not the desired surface
```

Line 1: Start the `for-each` loop.

Line 2: Definition of the `lambda` environment. The variable “`my-test`” holds the current item of the list.

Line 3: Definition of the local variable “`my-search-string`” with the value "`velocity-inlet`" as string

Line 4: Check if the current item of the list is identical with the value stored in `my-search-string`. The required data type conversion from symbols to strings is done by the format procedure.

Line 5: If previous check returns `#t`, display a success message.

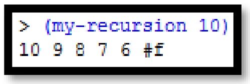
Line 6: If previous check returns `#f`, display a failure message

Line 7: Call the procedure that returns a list of all post-processing surfaces. Each item of the list is passed to the variable `my-test` (line 2) one by one. Note that you get an error if no case is loaded.

Recursion

Recursion is a powerful version of iterations which is commonly used. However, it is very easy to create infinite loops with it. Essentially, you define a procedure that calls itself.

```
001  (define my-recursion
002    (lambda (n)
003      (if (> n 5)
004        (begin
005          (display n) (display " ")
006          (my-recursion (- n 1))))))
```



```
> (my-recursion 10)
10 9 8 7 6 #f
```

Line 1: Define the name “`my-recursion`” for the procedure.

Line 2: Define a parameter “`n`” that has to be passed as argument when you call the procedure.

Line 3: Check if n > 5

Line 4: Execute the code in lines 5 and 6 if the condition of line 3 returns `#t`.

Line 5: Display the parameter followed by a white space

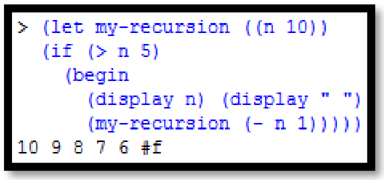
Line 6: Call the same procedure again. The argument is the current parameter reduced by one.

Note that you will get an error message if the argument you pass to the procedure is not a number.

Be very careful when using recursion. Test your procedure with small cases because it very easy to lock up Fluent in an infinite loop. If this happens, use the cleanup script that Fluent creates in its working directory to terminate the Fluent process.

Instead of defining procedures you can also use something called “named let”. The syntax is comparable to a procedure.

```
001  (let my-recursion ((n 10))
002    (if (> n 5)
003      (begin
004        (display n) (display " ")
005        (my-recursion (- n 1)))))
```



```
> (let my-recursion ((n 10))
  (if (> n 5)
    (begin
      (display n) (display " ")
      (my-recursion (- n 1)))))
10 9 8 7 6 #f
```

Line 1: Define the local name “`my-recursion`” together with the local variable `n` and its initial value

Line 2: Check if n > 5

Line 3: Execute the code in lines 5 and 6 if the condition of line 3 returns `#t`.

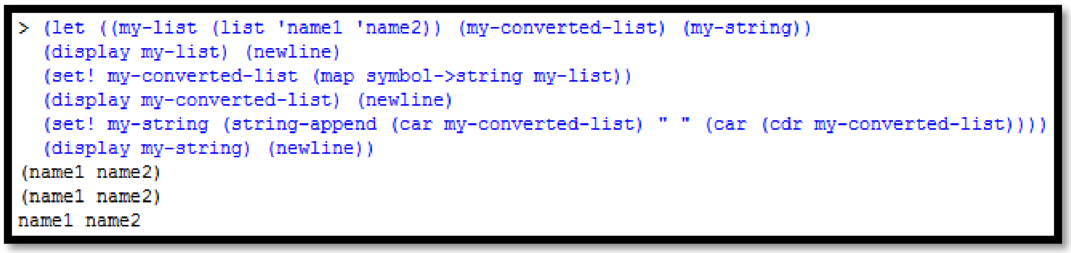
Line 4: Display the parameter followed by a white space

Line 5: Call the same procedure again. The argument is the current parameter reduced by one.

Mapping

You can apply a procedure on every element of a list with `map`. The return value is a list of the same dimension. Be careful that the list contains only valid elements before mapping. Error handling inside the procedure takes more time than doing it outside.

```
001  (let ((my-list (list 'name1 'name2)) (my-converted-list) (my-string))
002    (display my-list) (newline)
003    (set! my-converted-list (map symbol->string my-list))
004    (display my-converted-list) (newline)
005    (set! my-string (string-append (car my-converted-list) " " (car (cdr my-
     converted-list))))
006    (display my-string) (newline))
```



```
> (let ((my-list (list 'name1 'name2)) (my-converted-list) (my-string))
  (display my-list) (newline)
  (set! my-converted-list (map symbol->string my-list))
  (display my-converted-list) (newline)
  (set! my-string (string-append (car my-converted-list) " " (car (cdr my-converted-list))))
  (display my-string) (newline))
(name1 name2)
(name1 name2)
name1 name2
```

Line 1: Definition of three local variables. The variable “my-list” contains a list of two symbols. Fluent returns such lists for zones or domains. The other two variables are used later.

Line 2: Display the list as defined in line 1 and a line break

Line 3: The content of the second parentheses is the mapping procedure. It applies the data type conversion to all elements of “my-list”. Note that Fluent reports a wta error (wrong type of argument) if one item of the list is not a symbol. The result is a list that is stored in the variable “my-converted-list”.

Line4: Display the converted list and a line break. Note that the output in the Fluent console is identical to the output of line 2. Still, the data types are different.

Line 5: Append both items of the converted list to a single string. This could not be done with symbols.

Line 6: Display the string.

For this example, it would be possible to apply the data type conversion directly in the string-append procedure. But when you work with large lists, for example for automatic renaming of boundary conditions, mapping is much more efficient.

You are not limited to procedures that expect just one argument. Just be sure that you provide all required arguments within the parentheses of the map procedure.

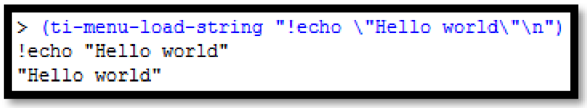
System commands

There are two ways to use system commands:

1. Use the TUI command to pass on system commands
2. Use a Scheme procedure

There is no recommendation which method should be preferred. The following examples are valid for Windows and Linux systems.

```
(ti-menu-load-string "!echo \"Hello world\"\n")
```

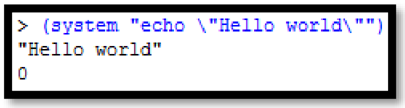


```
> (ti-menu-load-string "!echo \"Hello world\"\n")
!echo "Hello world"
"Hello world"
```

The procedure `ti-menu-load-string` passes the string to the Fluent text interface to be interpreted as text command. In this case the text command is the exclamation mark which sends the command to the command prompt (Windows) or the shell (Linux). Note that for Linux the output will not be visible in the Fluent console. Instead it is printed on the shell Fluent started from.

You can do the same with a shorter Scheme command.

```
(system "echo \"Hello world\"")
```



```
> (system "echo \"Hello world\"")
"Hello world"
0
```

When accessing system commands, you might need to check on which type of system you are. You can use `unix?` and `nt?` to check if the script runs on a Linux or Windows system, respectively.

```
(if (unix?)
  (system "kwrite")
  (system "notepad"))
```

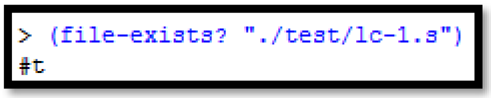
Accessing files

Checking if files exist

There is a simple Scheme command to check if a file exists. You can specify an absolute or a relative path with respect to the Fluent working directory.

```
(file-exists? "file-name")
```

For relative paths you can always use the Unix notation with the forward slash. Fluent can use this on Windows systems, too.

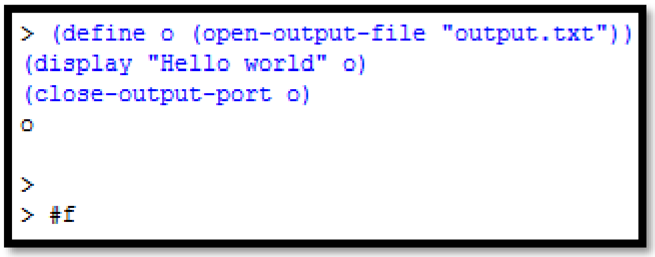


```
> (file-exists? "./test/lc-1.s")
#t
```

Writing files

To access files, you need to specify a port. This works just like defining a variable.

```
001  (define o (open-output-file "output.txt"))
002  (display "Hello world" o)
003  (close-output-port o)
```



```
> (define o (open-output-file "output.txt"))
(display "Hello world" o)
(close-output-port o)
o

>
> #f
```

Line 1: Define the file "`output.txt`" as output port. Access is possible with the symbol "`o`".

Line 2: Send the string "`Hello world`" to the file specified before.

Line 3: Close the port that the file can be accessed by other applications

Note that this will always overwrite the file if it exists already. If you want to append data, you can specify an additional Boolean value.

```
(define o (open-output-file "output.txt" #t))
```

You can check if a symbol is an output port:

```
(output-port? o)
```

After a port is closed with close-output-port you can no longer send data to the file. The port itself is still defined but you cannot access it any more. To re-use it, you would need to open it again with a `set!` Command.

Note: `(display "\n" o)` does not add the expected line break to a file.
You can use `(newline o)` instead.

Reading files

Similar to output ports you can define input ports.

```
(define i (open-input-file "input.txt"))
```

When done with the file you should close it just like output files.

```
(close-input-port i)
```

Of course you can also check if a symbol is an input port.

```
(input-port? i)
```

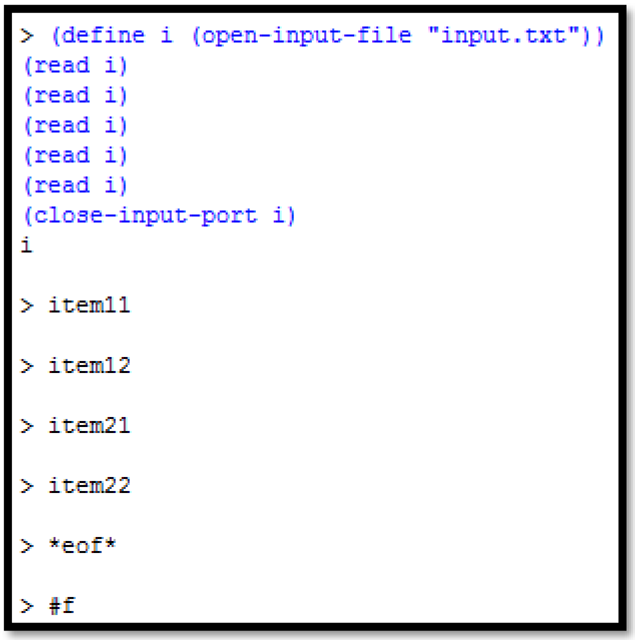
You can read an item from a file with `read`. This will read all characters to the next white space or line break.

Example input file called “input.txt”, located in the Fluent working directory:

```
item11 item12
item21 item22
```

These commands read five strings of the file which is everything this specific file contains.

```
(define i (open-input-file "input.txt"))
(read i)
(read i)
(read i)
(read i)
(read i)
(close-input-port i)
```

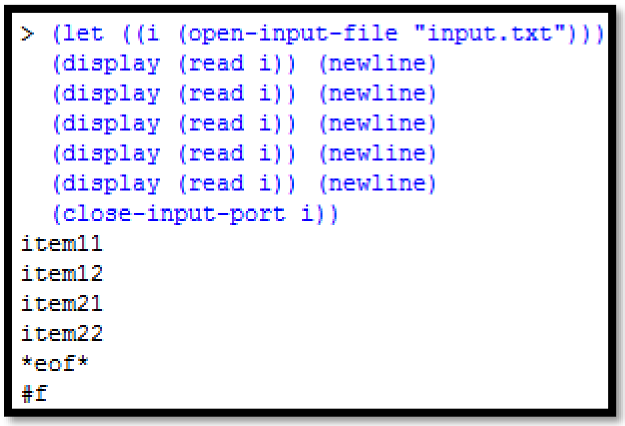


```
> (define i (open-input-file "input.txt"))
(read i)
(read i)
(read i)
(read i)
(read i)
(close-input-port i)
i

> item11

> item12

> item21

> item22

> *eof*

> #f
```

Notice the output `*eof*` which stands for end of file. You can use this in iterations to identify when there is nothing more to read.

You can also put everything inside a `let` statement for local definition.

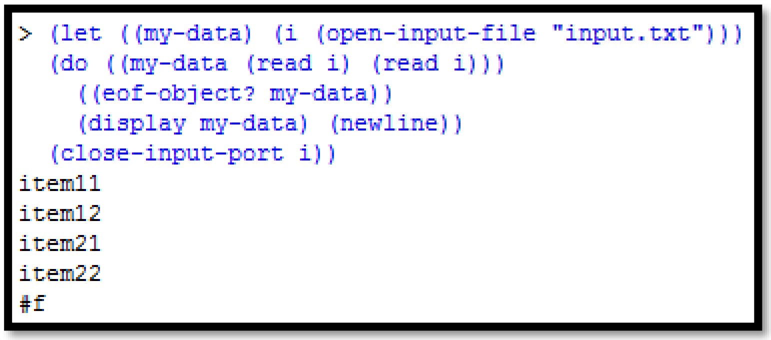
```
(let ((i (open-input-file "input.txt")))
  (display (read i)) (newline)
  (display (read i)) (newline)
  (display (read i)) (newline)
  (display (read i)) (newline)
  (display (read i)) (newline)
  (close-input-port i))
```



```
> (let ((i (open-input-file "input.txt")))
  (display (read i)) (newline)
  (display (read i)) (newline)
  (display (read i)) (newline)
  (display (read i)) (newline)
  (display (read i)) (newline)
  (close-input-port i))
item11
item12
item21
item22
*eof*
#f
```

Of course you can also use a loop through the file.

```
001 (let ((my-data) (i (open-input-file "input.txt")))
002   (do ((my-data (read i) (read i)))
003     ((eof-object? my-data))
004     (display my-data) (newline))
005   (close-input-port i))
```



```
> (let ((my-data) (i (open-input-file "input.txt")))
  (do ((my-data (read i) (read i)))
    ((eof-object? my-data))
    (display my-data) (newline))
  (close-input-port i))
item11
item12
item21
item22
#f
```

Line 1:	Define the local variables “`my-data`” and “`i`”. `i` is actually an input port, pointing to a text file
Line 2:	Loop through each string in the file that is separated with a white space. `(read i)` has to be used twice. The first statement is the initial value for the variable `my-data`, the second statement is executed for each loop.
Line 3:	When my-data is equal to `*eof*` the loop stops. No code is specified, so the script continues in line 5 (after the do loop) when the condition evaluates to `#t`. The output in the Fluent console if `#f` because there is no code specified for the case `#t`.
Line 4:	Display the value of `my-data` followed by a line break
Line 5:	Close the input port

Summary

Scheme offers you a versatile toolset to automate tasks in ANSYS Fluent. You can define procedures and add them to your .fluent file to have easy access to features you need all the time. You can also add menu items, create your own panels and even create user-define keyboard shortcuts to execute your scripts quickly.

After working through this document you should have the basics to develop your own scripts. Furthermore, you should be able to understand most of the examples that are available on the ANSYS Customer Portal. Search the solutions for the keyword “scm” and filter the results by Fluent to get relevant search results.

The huge downside of Scheme scripting is the lack of documentation and support. If you have trouble with your Fluent simulations, rename the .fluent file and try to reproduce the issues without using scripts.

Contact the ANSYS technical support only when you are sure that the problem exists without using Scheme.

If you need a *reliable* customization or scripting solution for your projects for a specific Fluent version, you can contact the ANSYS consulting team. Provide a detailed description of your requirements and ANSYS can make you an offer to develop a solution for you.

This document is provided for your convenience as self-help content. No support is provided for its content. The examples were tested with Fluent 17.0, 17.1 and a development version of Fluent 18. But ANSYS cannot guarantee that everything works with these or any other releases.

If you are interested to learn more about procedures that let you process output of text commands or return lists of zones or materials, you can continue with the second part of Introduction to Fluent scripting. You can find it in ANSYS solution 2042682 [8].

If you want easy access to your scripts with your own menu items, panels and keyboard shortcuts, check out part three of Introduction to Fluent scripting. You can find it in ANSYS solution 2042683 [9].

Index

References

[1] ANSYS, Inc., "ANSYS Fluent User's Guide, Release 17.1," ANSYS, Inc., Canonsburg, PA, 2016.

[2] ANSYS, Inc., "ANSYS Fluent Customization Manual, Release 17.1," ANSYS, Inc., 2016.

[3] D. Sitaram, "Teach Yourself Scheme in Fixnum Days," 1 June 2015. [Online]. Available: https://ds26gte.github.io/tyscheme/index.html.

[4] R. K. Dybvig, "The Scheme Programming Language, Fourth Edition," 2009. [Online]. Available: http://www.scheme.com/tspl4/.

[5] M. Javurek, „Fluent Scheme Dokumentation,“ Johannes Kepler University Linz, Institute of Fluid Mechanics and Heat Transfer, Linz, 2011.

[6] Massachusetts Institute of Technology, "MIT/GNU Scheme 9.2," 2014. [Online]. Available: https://www.gnu.org/software/mit-scheme/documentation/mit-scheme-ref/index.html.

[7] R. Kelsey, W. Clinger and J. Rees, "Revised Report on the Algorithmic Language Scheme," 20 February 1998. [Online]. Available: http://www.schemers.org/Documents/Standards/R5RS/HTML/r5rs.html.

[8] ANSYS, Inc., "Solution 2042682: Introduction to ANSYS Fluent scripting - Part 2 - Accessing solver data," ANSYS, Inc., 2016.

[9] ANSYS, Inc., "Solution 2042683: Introduction to ANSYS Fluent scripting - Part 3 - Creating keyboard shortcuts, menu items and panels," ANSYS, Inc., 2016.

Keywords: Best-Practice; best practices; guidelines; guideline; Scheme; Script; Scripting; ANSYS Fluent; Journal; Journaling; TUI; Text User Interface; Automation

Contributors:

- Akram Radwan, Senior Technical Support Engineer, ANSYS Germany, Customer Excellence, European Technology Group
- Dr.-Ing. Amine Ben Hadj Ali, Senior Technical Support Engineer, ANSYS Germany, Customer Excellence