F21MP
Masters Project and Dissertation

Script execution made simple with

SML/NJ

Due on Thursday 04 May 2023

Submitted by: Dayanandan Natarajan

Supervisor: Dr Joe Wells

School of Mathematical and Computer Sciences

Heriot-Watt University

Declaration of Authorship

I, Dayanandan Natarajan, declare that this thesis titled, 'Script execution made simple with SML/NJ' and the work presented in it is my own. I confirm that this work submitted for assessment is my own and is expressed in my own words. Any uses made within it of the works of other authors in any form (e.g., ideas, images, text, tables, programs) are properly acknowledged at any point of their use. A list of the references employed is included.

Signed: Dayanandan Natarajan

Date: 01/05/2023

Abstract

"Standard ML (SML) is a safe, modular, strict, functional, polymorphic programming language with compile-time type checking and type inference, garbage collection, exception handling, immutable data types and updatable references, abstract data types, and parametric modules". Developers of compilers, programmers, and others collaborating with theorem provers have used it to create formal proofs [1]. A free and open-source compiler called SML/NJ offers an SML programming environment. While it does not evaluate, SML/NJ processes the instructions in a manner like the well-known REPL (Read-Eval-Print-Loop) approach. It reads, parses, does type checks, compiles, links, runs, and modifies global environments. It loops back through the full cycle until all the instructions have been carried out. The term "script" in this article refers to a file containing source code which are also executable files. SML/NJ doesn't have the capability to execute a script. *smlnj-script* originally developed by project supervisor Dr Joe Wells, gave the capability to build SML scripts in a file and run like regular operating system files. Additional features of *smlnj-script* include support for Unicode and regular expressions. *smlnj-script* is a slightly modified version of SML/NJ which gives the capability to execute SML programs as a script. *smlnj-script* is written as a layer on top of SML/NJ. This dissertation aims to add and merge all *smlnj-script* features into SML/NJ and eliminate the need to maintain *smlnj-script* as a separate implementation. Students, programmers, and developers will benefit from the ability to write SML source code in a script and have them run like Python or Perl.

Acknowledgements

I would like to thank my supervisor Dr Joe Wells for his support, guidance and knowledge sharing on this topic.

List of Abbreviations

SML	Standard ML.
SML/NJ	Standard ML of New Jersey.
CM	Compilation and Library Manager (a component of SML/NJ).
JSON	JavaScript Object Notation.
REPL	Read-Eval-Print Loop (In SML world REPL is used to denote read, parse, type check, compile, link and execute.)
HWU	Heriot Watt University

Table of Contents

1. Introduction

1.1 Basic Concepts

This paragraph will aid readers in understanding the purpose. Most programming languages break down the execution of a program's source code into stages like compilation, linking, and execution. In contrast to certain other programming languages like Python and Perl, where the programme can be run straight from the source code, the user must conduct or provide instructions for each phase in the former languages. Additionally, in some programming languages, running a programme on the command line requires passing the instructions through a wrapper script, which necessitates the need of two files. One of the main goals of this project is to execute an SML programme in one step like a script.

1.2 Aim

SML is a procedural language that supports higher-order functions and abstraction. Developers of compilers, programmers, and others collaborating with theorem provers have used SML to create formal proofs [1]. Though SML/NJ (a free and open-source compiler) offers an SML programming environment, it doesn't evaluate and processes the instructions in a manner like the well-known REPL (Read-Eval-Print-Loop) approach. It reads, parses, does type checks, compiles, links, runs, and modifies global environments. It loops back through the full cycle until all the instructions have been carried out. SML/NJ doesn't have the capability to execute a script (file containing source code which are also executable files). *smlnj-script* originally developed by project supervisor Dr Joe Wells, gave the capability to build SML scripts in a file and run like regular operating system files. Additional features of *smlnj-scripts* include support for Unicode and regular expressions. *smlnj-script* is a slightly modified version of SML/NJ which gives the capability to execute SML source code as a script. *smlnj-script* is written as a layer on top of SML/NJ. This dissertation aims to add and merge all *smlnj-script* features into SML/NJ and eliminate the need to maintain *smlnj-script* as a separate implementation. Students, programmers, and developers will benefit from the ability to write SML source code in a script and have them run like Python or Perl.

The project's goal is to make it easier for college students to complete their class assignments so they can create and execute SML source code as a script.

Additionally, the project seeks to cooperate with SML/NJ implementors to include these new capabilities and have SML/NJ published in the future with these features.

1.3 Objectives

The SML-NJ code will be examined for suitable places where the *smlnj-script* features can be added or merged. Before merging with SML/NJ, *smlnj-script* features will be evaluated for compatibility with the most recent version of SML/NJ and will be modified to meet new standards. SML/NJ implementors will be given information on the new features, and we'll work to have them included in a future version.

1.4 Stakeholder

Below are the stakeholders of this work.

Stakeholders	Interests
Primary:	*Have direct relationship to the project*
Dayanandan Natarajan (myself)	Student working on this project.
Dr Joe Wells	Supervisor and original author of *smlnj-script*.
SML/NJ Implementors	They are the maintainers of SML/NJ and publish the SML/NJ upgrades.
Secondary:	*Have indirect relationship to the project*
Students	Beneficiaries of the resultant product.
Programmers	Beneficiaries of the resultant product.
Developers	Beneficiaries of the resultant product.

2. Literature Review

2.1 Standard ML (SML '90 & '97)

Standard ML '90 is a procedural language that supports higher-order functions and abstraction very strongly because it is highly typed. SML maintains fairly like the λ-calculus in terms of syntax and semantics. The language has undergone a minor change and simplification to become Standard ML '97. A new SML base library was included with the language definition to allow a variety of systems and applications programming. It has a large selection of pre-defined modules that include fundamental types, input/output capabilities, and interfaces for portable operating system interaction [6].

2.2 Implementations of SML

There are multiple variants of SML implementations available, from interpreters to byte-code compilers, to incremental compilers, to whole-program compilers. The two main capabilities which differ them are REPL and FFI. An interactive top-level read-eval-print-loop (REPL) capability enables users to enter specific top-level SML declarations, which are subsequently parsed, type verified, executed (either as machine code or as interpreted code), and the results are displayed. FFI (foreign function interface) enables calling of SML from other languages as well as functions in C libraries and other languages [7].

Some of the main implementations are,

- MLton
- SML/NJ
- Poly/ML
- SML#

For our dissertation we will be seeing more of SML/NJ.

2.3 SML/NJ

SML/NJ is an open-source compiler and programming environment for SML, which was developed at Bell Labs. It was initially created in collaboration between Princeton University and Bell Laboratories, and later joined by Yale University and

AT&T. The most recent version of SML/NJ implements the Standard ML language's SML '97 revision, including the standard basis library. The SML/NJ source code is provided without charge or warranty. In accordance with the relevant licence and copyright notice, anybody may use, copy, modify, and distribute the software [8].

Except for the runtime system, which is written in C, SML/NJ is primarily written in Standard ML. Numerous big systems have been implemented using it, particularly in the areas of applied logic and verification, programme analysis, and advanced compilers [8].

Some of the key features of the SML/NJ system are below [8],

a. An interactive top level based on incremental compilation is offered by SML/NJ.
b. SML/NJ uses Compilation Manager (from Matthias Blume), to simplify the development of large software projects.
c. A range of general-purpose data structures, algorithms, and utilities are offered by the SML/NJ library (such as finite sets and maps, regular expressions, pretty-printing). The other main SML implementations also use SML/NJ library and are periodically resynchronized. SML/NJ library not necessarily denoted as SML/NJ.
d. Higher-order functors, OR-patterns (logical "or" operations), first-class continuations, and other helpful features are added to the SML '97 language by SML/NJ.
    i. Higher-order *functor* is a module that take one or more other modules as a parameter [18][19].
    ii. First-class continuations are continuations used as values. Continuations can be the value of a variable, an argument for a function call, or the return value of a function. It can be considered as a function since it takes a value and produces a value [17].

e. A simple quote/anitquote mechanism offers support for manipulating "object languages". The command arguments contain the protected characters get passed along [20].

2.4 Programming process – Compile, Link and Execute

A program written in any programming language needs to be compiled and linked before execution. Programming languages vary in this process; they can be either compiled or interpreted based on how they are compiled, linked, and executed [15][16].

Compiled Programming Languages

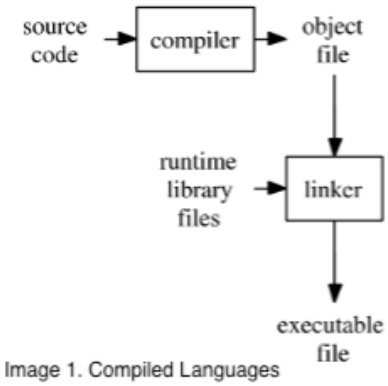

Image1: Compiled Languages [16].

Interpreted Programming Languages

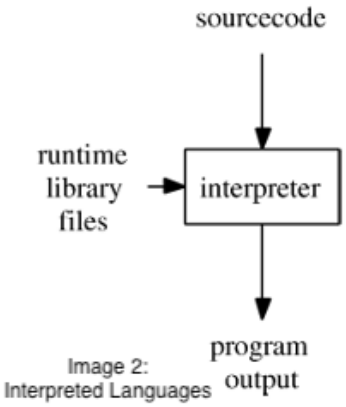

Image2: Interpreted Languages (ex: Python/Perl) [16].

Programming language like Java doesn't fit either in compiled or interpreted language, they have their own compiler which converts the source code into byte code which can be run on any machine using their own interpreter.

2.5 SML/NJ Compilation and Execution

When SML/NJ is started over the command line, it gets loaded and prompts to enter a top-level declaration through keyboard. When a top-level declaration is given to the prompt with a termination character semicolon (';'); SML/NJ understands the end of declaration and starts to process the declaration. "SML/NJ performs the following actions: elaborate (performs type checking and static analyses), compile (obtain executable machine code), execute, and finally print the result of this declaration" [22].

Rather than typing the program in top-level declarations it can be put together in a single file and loaded into SML/NJ system. SML/NJ processes the instructions using a method like the well-known REPL (Read-Eval-Print-Loop) methodology, though it does not evaluate. It reads, parses, does type checks, compiles, links, runs, and modifies global environments. It repeats the entire cycle in loops until all the instructions are processed. Once all the declarations in the program have been compiled and executed. Here comes an optional handy feature dumping, which creates a completely pre-compiled version of the program that can start-up faster.

At any given point of time, the snapshot of application's memory called as heap dump contains information in binary format, they contain information like objects in memory, their values, their size, and reference to other objects [23]. SML/NJ gives the capability to ship a program heap; upon loading a file in to SML/NJ compiler, it gets parsed, compiled, and executed. Dumping saves the system heap into a file. This file, which is a virtual memory image and not an executable, can be exported [22]. The exported heap can be executed by feeding the argument of the heap file to SML over a shell script.

Compilation and Library Manager (CM) is used for managing the compilation of SML programs and its required libraries. SML programs can be organized into separate modules with its own source file with CM. It also allows each module to be compiled independently, and compiled code to be linked together to form a complete program. Each module is compiled and linked in the proper order thanks to CM's management of inter module dependencies. This helps to prevent problems with circular dependencies and guarantees that the programme is successfully compiled and linked.

A library manager is another feature of the CM system that enables the management and usage of SML libraries in SML programmes. An assortment of SML modules that can be utilised in various SML programmes make up a library. With a separate source file for each module, libraries can be provided using the library manager in the same way that modules are. The library manager also handles the dependencies between modules in the library, ensuring that they are compiled and linked in the correct order. When a library is used in an SML program, the library manager automatically handles the loading and linking of the required library modules.

Overall, the CM system provides a powerful and flexible way to manage the compilation and linking of SML programs and libraries. By organizing programs and libraries into separate modules and managing the dependencies between them, CM helps to ensure that SML programs are correct, reliable, and maintainable [24]. CM gives the capability to load a specific file from the libraries which are precompiled. It can be loaded and added into the memory image whenever required.

To execute a SML program, the end-user needs to have two files, a program, and a wrapper script. The SML program will be run using the instructions in the wrapper script. Sometimes a long command line can be used as an alternative to a wrapper script. For example, to run a program 'helloworld.sml' there should be wrapper script 'helloworld.sh' available with instructions as below,

```
#!/bin/sh
sml /full/path/of/directories/to/helloworld.sml
```

2.6 smlnj-script

smlnj-script program is an extension to SML/NJ implementation. In SML/NJ, reading, parsing, type checking, compiling, linking, execution, modify global environments and looping happens in separate phases and the program is executed through instructions in a wrapper script. Unix has a special support for script; with a special instruction (program name preceded by a sharp exclamation) inside the script, the path that was initially used when attempting to start the script is passed to the interpreter programme that is given by the loader as an argument, allowing the programme to use the file as input data [14]. *smlnj-script* is an extension to SML/NJ and designed to perform all the operations in a single phase through a single file like

Python or Perl. It also includes run time machinery, with additional utility functions defined and a specialized start-up routine. Here are *smlnj-script* capabilities,

a. All the features of the SML language (as implemented by SML/NJ).
b. A portion of SML Basis Library that is provided by the SML/NJ implementation.
c. SML/NJ library.
d. New functions added as part of SmlnjScriptUtils *structure*.

2.7 What happens when *smlnj-script* gets loaded?

Whenever a user tries to run a script or load the *smlnj-script* directly, the SML/NJ gets preloaded, then it dumps the entire virtual memory image (of everything that's loaded in SML/NJ at that moment) into a heap file, converts the file into assembly code, gets assembled by an assembler into a binary file which can be executed at a later stage. Upon execution it recreates a memory image and starts it up again as *smlnj-script*.

2.8 Key features of *smlnj-script*

The *smlnj-script* has the following key features,

a. Execute as a script:

   The very first and most important feature of *smlnj-script* is the ability to run the sml source code as a script. It has the capability to recognise the script as a SML source code and process the instructions in the file by passing them through REPL.

b. Silencing the Compiler:

   Running a script through *smlnj-script* goes thru this redevelopment loop, cycle of read, parse, type check, compile, links, execute and modify global environment. The compiler produces bunch of output messages printed for each operation which is going to confuse the user especially when there is a compilation error. silenceCompiler function gives the capability to turns off these compiler messages like print of the name, type, and value of all

identifiers that get added to the top-level environment, autoloading and compilation error messages. This completely silences the compiler and hides type error messages. There is no option to turn off the messages while keeping the error messages [10]. silenceCompiler function intercept the compiler messages, accumulates these messages, and stores them in a variable to be printed to the user at a later stage when there is a compilation error or program crashes.

```
val silenceCompiler : unit -> unit =
SmlnjScriptUtils.silenceCompiler
```

c. Overloading toString function:

SML/NJ has the limitation on overloading toString operator especially with complex types like constructors with arguments. To print out a list of integers or real numbers or float data types a custom-built function is needed. The toString overloading function needs further improvement and advancements to support string functions.

The toString (and %) overloaded operator:

```
val % : 'a -> string = ...
val toString : 'a -> string = ...
```

The % and toString operators are both overloaded operators to convert many types into strings. They are equivalent to Int.toString, Bool.toString, Real.toString, etc. There is a room for improvement to add support for more types, provided the type is “atomic” with SmlnjScriptUtils.extendToString [10].

d. The SmlnjScriptUtils structure:

The SmlnjScriptUtils *structure*, denoted by the short name ‘U’, is a collection of functions with its own features.

```
structure SmlnjScriptUtils = ...
structure U = SmlnjScriptUtils
```

Some of those functions are,

i. The raisePrintingLimitsToMax function:

```
val raisePrintingLimitsToMax : unit -> unit = ...
```

The function is very useful while debugging and the default limits cause too much of the data structures to be truncated when printed by the compiler [10].

ii. The interact and continue functions:

```
val interact : unit -> unit = ...
val continue : unit -> unit = ...
```

The function interact will interrupt the script and transfers control to the user who is at the keyboard, and function continue resumes execution by returning from a call to interact [10].

iii. The extendToString function

```
val extendToString : string -> unit = ...
```

The function extendToString extends the top-level overloaded operators toString and % with an additional case. Supply the name of a function of type T -> string where T is an “atomic” type. The type T is atomic when it is a type name without arguments and there is no definition available in scope that allows the compiler to deduce it is equal to a non-atomic type [10].

iv. The evalString and useString functions:

```
val evalString : string -> unit = ...
val useString : string -> unit = ...
```

Both evalString and useString enter into a loop of parsing the string to find a prefix that is a top-level SML form, type checking and compiling this form, and executing the resulting machine code, and then repeating with the rest of the string. The difference is that evalString throws any new top-level definitions away when it is done and returns (only their side-effects persist) while useString extends the top-level environment with the new definitions when it returns. In both cases new top-level definitions are available for remaining forms processed during the call to evalString or useString [10].

v. The StringSet and StringMap *structures*:

```
structure StringSet : ORD_SET = ...
structure StringMap : ORD_MAP = ...
```

The *structures* StringSet and StringMap provide implementations of sets and finite maps (finite maps are sometimes called “dictionaries” or “associative arrays”) for the type “string”. The operations available are defined in the ORD_SET and ORD_MAP *signatures*, which are part of the SML/NJ library [10].

vi. The q and qq quoting functions:

```
val q = SmlnjScriptUtils.q
val qq = SmlnjScriptUtils.qq
```

The functions q and qq are abbreviations for things in SmlnjScriptUtils, giving you a compact syntax for writing

strings with stuff spliced into a string template. These operators are intended to be used with SML/NJ's quasiquote syntax to make quoting operators [10]. The quasiquote syntax is also supported by Moscow ML and the ML Kit, but is not supported by some other major SML implementations like MLton and Poly/ML.

To illustrate, here are some expressions that evaluate to true:

```
let val x = 7 in q`The value of x is ^(% x)!` =
"The value of x is 7!" end
let val y = "Jack" in qq`My name is ^y.\n` = "My
name is Jack.\n" end
```

In the above code, the use of '% x' is a use of toString feature. Both 'q' and '%' are two independent features, combined together they give the capability to print a string in middle of a string or include value of a variable. The difference between q and qq is that qq interprets SML-style escape syntax in the literal text while q does not. For example, this evaluates to true:

```
q`This is not a newline: \n` = "This is not a
newline: \\n"
```

The names q and qq are loosely inspired by Perl's q and qq operators [10].

e. Dumping and Creating executable:

SML/NJ gives the ability to dump and load back contents of virtual memory in the form of heap image into a file. When a SML is started again (later), by passing a special command line instructions through another script, the contents of the memory can be discarded and loaded with the contents of the heap image. *smlnj-script* uses SML/NJ's built-in utility functions *exportML* to save the heap image (current state of SML/NJ) in a file [22], *exportFN* to save the current state in the form of a function (function upon restarting takes in shell command-line arguments) in a file [22], and

'heap2exec' to convert heap image into an executable which can be run faster. 'heap2exec' wraps the binary SML/NJ runtime image and converts heap image into one executable. It makes use of built-in utility 'heap2asm' to build a stand-alone executable from the image [25], which can be used at a later stage.

f. CM Autoloading:

Some of the libraries which come with SML/NJ as part of Compilation and Library Manager are not loaded initially. It needs some expertise to load these libraries which is unusual for an end user. *smlnj-script* loads such libraries and make them available to the user, one of the common is regular expression library. This eliminates the need for end user's knowledge on CM.autoload.

g. Parse, Eval & Format:

Parse, Eval & Format gives the capability to make use of compilers ability to pretty print data which it has for implementing the redevelopment loop. In the last phase of the loop the results are printed out, SML/NJ has built-in ability to print out results of many types though they are not ordinarily available to the end users and programmers. At the end of each iteration loop this stores the value of the expressions which need to be printed out in a variable. It ignores the irrelevant part and keeps only the relevant part. This helps to convert the data of many types to strings.

h. Parser Combinators:

Parser Combinators gives the capability to operate on streams of arbitrary types rather than just streams of chars. This allows to implement Unicode support using the utf8 encoding.

i. UTF8 processing:

This module provides a structure with types and utility functions to work with Unicode characters written in utf8. It also provides structure with definitions which are needed during bootstrap and normal use.

This gives the ability to create stream of Unicode characters from a stream of bytes (old style ASCII characters).

j. Process function:

smlnj-script gives a value-added interface on top of the subprocess handling available in SML/NJ.

k. XML Wrapping:

This module is derived from SML XML library and work of Tom Murphy VII, which is again a wrapper of 'fxp' [27]. It is used to parse XML files.

l. Utility functions:

smlnj-script have utility functions to handle basic data operations and load required libraries. Structures String and CharVector have good overlap, some members in CharVector don't have no equivalents in String and the same goes for Substring and CharVectorSlice. The function is customized to give the capability of string repetitions, concatenations, comparison, and conversion operations. One of the capabilities is to auto load JSON library, so that end user need not to load the JSON library.

2.9 How to run a script?

In UNIX, to execute a file or script, the script should start with a hashbang ('#!') character followed by the path to the interpreter and any optional argument as follows,

```
#!PATH-TO-INTERPRETER OPTIONAL-ARGUMENT
```

and followed by instructions in the subsequent lines. The script also need execute permission. Section 2.13 will explain in detail on what happens when a script is executed.

2.10 How to run *smlnj-script?*

smlnj-script file also follows the same rule as mentioned in 2.9. Let's see a sample script 'HellolWorld' that runs *smlnj-script*, the script will start with the below statement followed by instructions in the subsequent lines.

```
#!/usr/bin/env smlnj-script P
```

When the above script is executed as below,

./HelloWorkd

the operating system will run the '/usr/bin/env' program with the arguments 'smlnj-script' and P.

The Interpreter can be invoked directly passing it the name of the script as its first argument. For example, to run a script 'HelloWorld' with three arguments arg1, arg2 and arg3, the interpreter can be invoked as below,

```
smlnj-script HelloWorld arg1 arg2 arg3
```

The script 'HelloWorld' doesn't need to have 'execute' permission if it's going to be passed as argument to interpreter.

2.11 How to work with SML/NJ?

Users can invoke SML/NJ by typing the key word 'SML' over command prompt in Windows or terminal in Linux, which will load SML in interaction mode,

```
$ sml
Standard ML of New Jersey (32-bit) v110.99.3 [built: Sun Apr 16
18:42:54 2023]
-
```

Users can break the interactive mode and come out of SML/NJ by issuing the keyboard shortcut 'Ctrl+D' or by issuing the SML command (OS.Process.exit OS.Process.success).

```
$ sml
Standard ML of New Jersey (32-bit) v110.99.3 [built: Sun Apr 16
18:42:54 2023]
```

```
- ^D
$
```

Users can type in the instructions line by line with the delimiter semicolon (';') at the end of line. The semicolon denotes the end of line and instructs the compiler to process the line thru REPL.

```
$ sml
Standard ML of New Jersey (32-bit) v110.99.3 [built: Sun Apr 16
18:42:54 2023]
- val x = "Hello World\n";
val x = "Hello World\n" : string
- print x;
Hello World
val it = () : unit
-
```

If an instruction is not delimited by a semicolon, the compiler will assume the instructions are continuing in next line and wait for a semicolon to mark the end of the instruction and submit it for REPL.

```
- val x = "Hello World\n"
= ;
val x = "Hello World\n" : string
- print
= x
= ;
Hello World
val it = () : unit
-
```

2.12 What happens when SML/NJ is loaded?

Now we know when a user invokes SML/NJ, an interactive system gets loaded and available to the user so that the instructions are keyed-in line by line to get

processed. Let’s go little deep into what happens after the SML/NJ is invoked and before the interactive terminal is presented to the user.

SML/NJ is built on both C and SML language. The key functions to interact with the operating system are written in C and SML is used to write the additional functions on top of it. If we look closely into the installation log of SML/NJ, we can understand this, all the C programs will be compiled and loaded first to create a run file and followed by all SML code parsed, compiled and loaded (which generates a heap image). The run file made of C binaries is an executable which takes the heap image as argument to load the SML.

Log from terminal when SML is invoked,

```
$ sml
CMD=`basename "$0"`
basename "$0"
++ basename /usr/local/smlnj/bin/sml
+ CMD=sml
…
…
exec "$RUN" @SMLcmdname="$0" "$HEAP" $ALLOC "$@"
+ exec /usr/local/smlnj/bin/.run/run.x86-darwin
@SMLcmdname=/usr/local/smlnj/bin/sml
@SMLload=/usr/local/smlnj/bin/.heap/sml
Standard ML of New Jersey (32-bit) v110.99.3 [built: Sun Apr 16
18:42:54 2023]
-
```

C executables starts the boot process which creates a boot environment and loads default basic functions creating a pervasive environment for the interactive system. When the boot environment is loaded, it looks for any command line arguments supplied by user and if there is any, it will be verified and validated against the pre-defined values and appropriate action is performed. The pervasive environment is binded with basic functions like arithmetic operations, string operations, infix binding of symbols and overloaded bindings that use overloaded declarations. When the interactive system is made available to the user, not all the structures are pre-loaded, only the structures which contains the basic functions are

loaded. Not all the structures are loaded to minimize the usage of memory and to optimize the performance. Auto-loading process helps to load the necessary libraries in the pervasive environment. A select few important libraries are set up to automatically load and are made available in the top-level environment as needed.

2.13 What happens when a script is executed?

In this section we will discuss about what happens when a script is executed over command prompt or terminal in Unix machine. When a script is executed, the shell interpreter will verify for execute permissions available on the script and if so, it will proceed with identifying the interpreter to be used. Once the interpreter is identified it continues to read and execute the instructions in the file.

Example:

```
$ ./samplescript
or
$ ./samplescript.sh
```

In case the execute permission is not available on the file, it be can be given using ‘chmod’ command over the command prompt or terminal,

Example:

```
$ chmod +x samplescript
$ ./samplescript
or
$ chmod +x samplescript.sh
$ ./samplescript.sh
```

From section 2.9 we know that the scripts in Unix start with ‘#!’, the character sequence commonly referred as ‘hashbang’ (also called as ‘sharp-exclamation’ and ‘shebang’). When a file starting with a hashbang is executed over operating system command prompt, the program loader parses the first line as an interpreter directive [26]. The program loader will execute the specified interpreter along with the path of the script passed as an argument to the interpreter.

For example, if a script starts as below,

```
#!/usr/bin/env perl
```

The script will be executed using the env command with the user's $PATH environment variable for the specified interpreter, in this case its Perl. Any additional arguments specified in the script first line will also be passed along to the interpreter.

For example, if a script starts as below,

```
#!/usr/bin/env -Ssml –arg1
```

The script will be executed using the env command with the user's $PATH environment variable for the specified interpreter. The -S option is used to pass the interpreter as an argument to the env command. The 'sml' argument following the -S option specifies the interpreter to use. In this case, the interpreter being used to execute the script is SML. Any argument following the interpreter will be passed on as additional argument to the interpreter. In this case, the argument 'arg1' is passed to the SML interpreter. If there are more additional arguments, they all will be passed along followed by end the path of the script.

2.14 What to be updated in SML/NJ?

As we know from the main objective of this dissertation, SML doesn't have the feature where a script with SML source code can't be executed. In other terms a script with SML source code can't be executed even if SML interpreter is passed as argument to env. Logically the script gets executed by the operating system but unsuccessful because SML doesn't have the capability to receive the arguments passed from program loader when a script is executed. SML does have the capability to receive the arguments if the interpreter is invoked directly over the terminal or command prompt.

For example, SML can take arguments like below,

```
$ SML –h
$ SML –H
```

```
$ SML –S
$ SML sample.sml
```

For example, the below scripts will fail,

```
#!/usr/bin/env –Ssml
(or)
#!/usr/bin/env -Ssml --script
```

Though SML interpreter is identified and passed with argument '–script', the argument will fail the interpreter since it won't be recognised by the interpreter. This is one of the reason *smlnj-script* is created as a layer on top of SML/NJ and made it easy for *smlnj-script* to give this capability to execute a SML source file as a script. Along with this it also featured other capabilities as detailed in section 2.6.

This is the most important and key change that need to be incorporated in SML/NJ to accept or recognise the new argument '—script' and able to process the instructions in the script. Implementation section will cover all the change information in detail.

3. Requirements Analysis

3.1 Functional Requirements

The following are the functional requirements originally planned to be analyzed, built, tested, and delivered.

No.	Requirement Description	MoSCoW	Priority
1	Identify the right module and merge *smlnj-script*'s execute as a script capability into SML/NJ.	M	High
2	Identify the right module and merge *smlnj-script*'s SilenceCompiler function into SML/NJ.	M	High
3	Identify the right module and merge *smlnj-script*'s toString overloading capability into SML/NJ.	M	High
4	Identify the right module and merge *smlnj-script*'s CM Autoloading capability into SML/NJ.	M	High
5	Identify the right module and merge *smlnj-script*'s Parse, Eval & Format functions into SML/NJ.	M	High
6	Identify the right module and merge *smlnj-script*'s Parser Combinators capability into SML/NJ.	M	Medium
7	Identify the right module and merge *smlnj-script*'s Utf8 processing capability into SML/NJ.	M	Medium
8	Identify the right module and merge *smlnj-script*'s Process functions into SML/NJ.	S/C	Low
9	Identify the right module and merge *smlnj-script*'s XML Wrapping capability into SML/NJ.	S/C	Low
10	Identify the right module and merge *smlnj-script*'s Utility functions into SML/NJ.	S/C	Low

3.2 Non-Functional Requirements

The following non-functional requirements are planned to be analyzed, built, tested, and delivered.

No.	Requirement Description	MoSCoW	Priority
1	Usability – End users, programmers, developers, and students should be able to use this final product to run the scripts with minimal effort.	M	High
2	Stability – The final product should be stable enough to run the scripts without getting crashed.	M	High
3	Reliability – The final product should be capable to producing accurate results.	M	High

4. Methodology

This dissertation's major objective is to incorporate *smlnj-script* capabilities into SML/NJ and to do away with the necessity to maintain *smlnj-script* as a separate addition to SML/NJ. To determine where to incorporate *smlnj-script* capabilities, the SML/NJ core code need be examined. It is necessary to evaluate and investigate SML/NJ's Compilation and Library Manager to discover the proper incantations and make advantage of its capacity to carry out efficient auto-loading without requiring the script writer to possess in-depth knowledge of SML/NJ. Before merging with SML/NJ, *smlnj-script* features will be evaluated for compatibility with the most recent version of SML/NJ and will be modified to meet new standards.

The project's goal is to introduce new features to SML/NJ implementors so they can be included in a later release. In the event of a redistribution, the initiative will assist licensing agreements between HWU and SML/NJ implementors. We anticipate that HWU will either unilaterally license the project code under the same conditions as SML/NJ or will assign its copyright interest in *smlnj-script* to "The Fellowship of SML/NJ," with either course of action being discussed with SML/NJ maintainers before moving forward.

5. Professional, Legal, Ethical, and Social issues

5.1 Professional Issues

- This project conforms to the terms and conditions of the SML/NJ licence and use of all software and technologies provided by HWU in line with their terms of licence and terms and conditions.
- If SML/NJ implementors agree to the change, this project will be included back into SML/NJ with the appropriate licencing approval from Dr Joe Wells and HWU for utilising *smlnj-script* features.
- Any open-source software from a third party that is utilised will receive due acknowledgement.

5.2 Legal Issues

- This project adheres to HWU's licencing policies, and any source code protected by intellectual property rights is reused.
- If an agreement between HWU and SML/NJ implementors is necessary to add project features to SML/NJ, it will be arranged. If the agreement is unsuccessful or if either party is unwilling, the project shall remain the property of the author and HWU.
- For any exploitation and dissemination of third-party content, formal approval from the contributors will be obtained.

5.3 Ethical Issues

- No human survey has been done or will be done; this project is a technical implementation on top of an existing open-source tool; it does not involve any sensitive personal data of an individual. Only source code and author information are gathered for this project.

5.4 Social Issues

- This project stores source code and compiler information, as well as personally identifiable information like author names, which are both preserved and cited in the project papers.

6. Project Risk Assessment

The following risks have been identified, impact analysed, and mitigated.

Risk	Likelihood	Impact	Mitigation
Author's medical emergency.	Medium	High	Work with supervisor and University for extension or resubmission.
Supervisor's medical emergency.	Medium	High	Work with University and personal tutor for additional support.
Loss of project artifacts.	Low	Medium	Periodic backup will be taken in GitHub and University locations.
SML/NJ releases a new update.	Low	Low	Features that are planned to merge with SML/NJ will be modified to adapt to latest update from SML/NJ.
Unavailability of technical information	High	High	Reach out to SML/NJ implementors for details.
Project gets delayed and overrun planned timeframe.	Medium	Medium	High priority requirements will take precedence.
Software not compatible with Author's hardware.	Low	Low	University resources will be utilized.

7. Implementation

7.1 Software and Hardware used

SML/NJ version 110.99.3 (32 bit) was used on an Intel based MacBook pro with macOS 10.13.16. Though SML has latest versions 2021.1 and 2022.2 with 64-bit support, we encountered an issue with structure Backend which is recognised by SML but unable to load it and its associated structures and functions in interaction mode. This was a known issue with SML 110.99.3 (64-bit) and we got this confirmed with SML implementors. We finally decided to use 110.99.3 32-bit version of SML since 32-bit support stopped with 110.99.3.

Default SML implementation doesn't come with source code. SML must be manually installed with the support for source code update and recompilation. Once the installation files are downloaded from SML/NJ download site, the targets file in *config* folder need to be updated to request for source code before installation. This will enable the capability to update the source code, rebuild and reinstall SML.

7.2 Where to be updated in SML/NJ

Following files in the SML/NJ library were amended to include the *smlnj-script* capabilities in to SML/NJ,

- smlnj/base/cm/main/cm-boot.sml
- smlnj/base/system/smlnj/internal/boot-env-fn.sml
- smlnj/base/compiler/TopLevel/interact/interact.sig
- smlnj/base/compiler/TopLevel/interact/interact.sml
- smlnj/base/compiler/TopLevel/backend/backend-fn.sml
- smlnj/base/compiler/TopLevel/backend/backend.sig
- smlnj/base/compiler/TopLevel/interact/mutecompiler.sig (New component)
- smlnj/base/compiler/TopLevel/interact/mutecompiler.sml (New component)
- smlnj/base/compiler/INDEX
- smlnj/base/compiler/MAP
- smlnj/base/compiler/core.cm

7.3 Change details

The following section will explain the changes that are made in SML/NJ to add the features of *smlnj-script*,

7.3.1 cm-boot.sml

cm-boot.sml is the very first file that is loaded when the CM system is invoked. It contains the basic definitions and functions that are needed to initialize the system. Some of the key structures and functions are defined in this file. Structure CM is defined with main functions like CM.make, CM.recompile, and CM.sources to manage the compilation process. Functor LinkCM (which is linked with Structure Backend) is defined with key functions and one such is to process the command-line arguments passed by user through functions ‘args’ and ‘carg’.

Based on the command-line arguments passed, appropriate functions are invoked. From the command prompt or terminal along with sml interpreter if a file is passed as an argument, its passed thru function ‘processFile’ for further processing. In ‘processFile’, the file extension is evaluated for and put through another function ‘useFile’ for extensions ‘sml’, ‘sig’ and ‘fun’, and function make for extension ‘cm’.

In addition to that it also defines the basic error handling and exception handling functions that are used throughout the CM system. These functions are used to handle errors and exceptions that occur during the compilation process in a more efficient way. cm-boot.sml plays a vital role in initializing the CM system and providing the basic functionality that is needed for managing the compilation of SML programs.

The following changes were made in cm-boot.sml to recognise the new command line parameter passed from script.

a) ‘init’ function is the point of entry for all the functionalities defined in cm-bool.sml. It’s been called with other key functions imported from other structures as arguments. A new function ‘useScriptFile’ added in Structure Interact is been added as an additional argument to ‘init’ function. Purpose and usage of ‘useScriptFile’ will be detailed in the upcoming sections.

```
fun init (bootdir, de, er, useStream, useScriptFile, useFile,
errorwrap, icm) = let …
```

b) Function ‘procCmdLine’ is one of the key functions in cm-boot.sml which process the command-line instructions and arguments. A new function ‘processFileScript’ is added as a sub function to ‘procCMdLine’. ‘processFileScript’ takes the script file name passed as argument, performs a validation through function ‘checkSharpbang’, consumes the contents of first line through function ‘eatuntilnewline’ and calls the function ‘useScriptFile’ to process the script. The ‘useScriptFile’ function is called with file name and its content in the form of stream as arguments.

Source code:

```
fun processFileScript (fname) = let
    val stream = TextIO.openIn fname
    val isscript = checkSharpbang stream
    in
        if (isscript) = false
            then    ( Say.say [ "!* Script file doesn't start
            with #!. \n" ] )
            else    ( useScriptFile (fname, stream) )
    end
```

c) Function ‘checkSharpbang’ takes file stream as an argument, which is called with the script file contents as a stream from the calling function. The function is used to verify whether the first line of the stream starts with ‘#!’. In that case the function returns a Boolean response ‘true’ and if the stream doesn’t start with ‘#!’ then the function returns the Boolean response ‘false’.

Source code:

```
fun checkSharpbang (instream : TextIO.instream): bool = let
                val c = TextIO.input1 instream
                in
                    case c of
                        SOME #"#" => (
                            case TextIO.lookahead instream of
```

```
                                        SOME #"!" => eatuntilnewline
    instream
                                        | SOME c => false
                                        | NONE => false
                                        )
                                | SOME c => false
                                | NONE => false
                        end
```

d) If the file that come in as an argument to function ‘processFileScript’ is confirmed as script by function ‘checkSharpbang’, then the contents of first line of the script until the new line character ‘\n’ is consumed since the REPL won’t be able to process the first line. The first line of the script starting with ‘#!’ is more for the operating system to understand and SML won’t be able to understand the instructions. To consume the first line of the script function ‘eatuntilnewline’ is called with the script file stream passed as an argument.

e) Function ‘eatuntilnewline’ will consume all the characters in the script first line until a new line ‘\n’ character is encountered. Once the new line character is reached the function returns Boolean value ‘true’ to the calling function, which at that point will have the file stream stripped with first line, passes the file stream for further processing.

Source code:

```
fun eatuntilnewline (instream : TextIO.instream): bool = let
    val c = TextIO.input1 instream
    in
        case TextIO.lookahead instream of
            SOME #"\n" => true
            | SOME c => eatuntilnewline instream
            | NONE => false
    end
```

f) SML can take in three type of arguments ‘rtsargs’ (Runtime system arguments), ‘options’ and ‘files’. ‘rtsargs’ are used to load specific information from SML Basic Library. ‘options’ are used either to set CM variables or to invoke help or to change any settings. ‘files’ are used either to load instructions to the top level

environment through (.sml/.sig/.fun) files or load custom libraries through .cm files.

Function ‘args’ is one of the key function in functor ‘LinkCM’ which validates the command line arguments. When a valid argument is passed from the command prompt or terminal, appropriate task is performed by calling the respective functions. Only ‘options’ and ‘files’ arguments are handled by function ‘args’. ‘rtsargs’ are handled separate in a different functor which is discussed later in this document. The arguments will be looked for whether a standalone argument or the following argument in the list is a supplementary argument to previous argument. Standalone arguments referred here are the ones which are used to perform a specific task without the need for next argument, for example passing ‘-h’ or ‘-H’ for help in command prompt. Consider another example ‘-type f’ where the first argument need the second argument to be present, this is what I referred here as supplementary argument. If the arguments are not in the predefined list in function ‘args’ or standalone argument, the control is transferred to function ‘carg’. Only the first two characters of the argument is passed to function ‘carg’, the assumption here is only the standalone arguments are passed. Function ‘carg’ does almost the same as function ‘args’, validates the argument passed and appropriate task is performed by calling its corresponding functions.

Function ‘args’ is updated to recognise the new argument ‘--script’ which is passed by the program loader from the script. Whenever ‘—script’ is passed as an argument, a new function ‘nextargscript’ is called.

Source code:

```
        fun args ([], _) = ()
          | args ("-a" :: _, _) = nextarg autoload'
          | args ("-m" :: _, _) = nextarg make'
          | args (["-H"], _) = (help NONE; quit ())
          | args ("-H" :: _ :: _, mk) = (help NONE; nextarg mk)
          | args (["-S"], _) = (showcur NONE; quit ())
          | args ("-S" :: _ :: _, mk) = (showcur NONE; nextarg mk)
          | args (["-E"], _) = (show_envvars NONE; quit ())
          | args ("-E" :: _ :: _, mk) = (show_envvars NONE; nextarg
mk)
          | args ("--script" :: _, _) = (nextargscript ())  (* line
added by DAYA *)
```

```
    | args ("@CMbuild" :: rest, _) = mlbuild rest
    | args (["@CMredump", heapfile], _) = redump_heap heapfile
    | args (f :: rest, mk) =
    (carg (String.substring (f, 0, 2)
       handle General.Subscript => "",
       f, mk, List.null rest);
     nextarg mk)
and nextarg mk =
let val l = SMLofNJ.getArgs ()
in SMLofNJ.shiftArgs (); args (l, mk)
end
```

g) A new function ‘nextargscript’ is defined to look for the remaining values in the arguments list, retrieve the head of the remaining argument’s list (which is expected to be the file name of the script) and pass the file name as an argument to another new function ‘processFileScript’.

The general assumption here is, first line in the script starting with ‘#!’ doesn’t have any argument passed after ‘—script’. In case of multiple argument to be passed, ‘—script’ will be last one in the list. The reason behind is, in the script when option ‘-S’ is used, it passes the interpreter as an argument to the ‘env’ followed by all other entries as arguments to the interpreter and finally the name of the script passed as the last argument to the interpreter. This means, the name of the script will be in the argument list immediately after entry ‘--script’. When ‘—script’ is passed from script, it gets picked by function ‘args’ and function ‘nextargscript’ is called to retrieve the script file name and call the function ‘processFileScript’ with the file name. Upon completion of function ‘processFileScript’, the process gets terminated and returns control to the command prompt or terminal.

Source code:

```
and nextargscript () =
        let val l = SMLofNJ.getArgs ()
        in SMLofNJ.shiftArgs (); processFileScript (hd l); quit ()
        end
```

7.3.2 boot-env-fn.sml

boot-env-fn.sml hosts the signature 'BOOTENV' and functor 'BootEnvF', they are the key components in creating the bootstrap environment. Key functions that are required to initiate the pervasive environment are declared here for importing. Function 'bootArgs' plays a key role in the bootstrap environment process, it handles the runtime system arguments passed from command line. These runtime arguments starting with '@SML*' are validated here and appropriate function is performed. Functions that are passed as arguments to the 'init' function in cm-boot.sml are declared here with their arguments data type and return data type.

In functor BootEnvF, function 'cminit' declaration is amended to include the argument and return data type of newly added function 'useScriptFile'.

Source code:

```
functor BootEnvF (datatype envrequest = AUTOLOAD | BARE
          val architecture: string
          val cminit : string * DynamicEnv.env * envrequest
                   * (TextIO.instream -> unit)(* useStream *)
                   * (string * TextIO.instream -> unit) (*
useScriptFile *)
                   * (string -> unit) (* useFile *)
                   * ((string -> unit) -> (string -> unit))
                                      (* errorwrap *)
                   * ({ manageImport:
                      Ast.dec * EnvRef.envref -> unit,
                    managePrint:
                      Symbol.symbol * EnvRef.envref -> unit,
                    getPending : unit -> Symbol.symbol list }
                  -> unit)
                   -> (unit -> unit) option
          val cmbmake: string * bool -> unit) :> BOOTENV =
struct

fun useFile f = if Backend.Interact.useFile f
          then ()
          else OS.Process.exit OS.Process.failure
        in
          U.pStruct := U.NILrde;
```

```
            cminit (bootdir, de, er,
                Backend.Interact.useStream,
                Backend.Interact.useScriptFile, (* added as part
of Execute as a script change *)
                errorwrap false useFile,
                errorwrap true,
                Backend.Interact.installCompManagers)
        end
```

7.3.3 interact.sig & interact.sml

One of the key structure in SML is Backend, it encapsulates the low-level details of the compilation and interpretation process. It allows the compiler and interpreter to operate at the higher level of abstraction. Interact is a sub structure with in Backend which provides functions to perform interactive input and output operations over a command prompt or terminal environment. Some of the key functions of Interact are 'useFile' (which takes SML source in a file and add those declarations in the file to the top level environment) 'useStream' and 'evalStream' (which takes SML source in the form of stream and add those declarations in the stream to the top level environment) and 'withErrorHandling' (which handles the error). A new function (useScriptFile) is added to Backend.Interact structure, which takes the file name and its content as a stream and process the stream by passing it to EvalLoop.evalStream.

a) New function declaration is added to interact.sig,

Source code:

```
val useStream : TextIO.instream -> unit
val useScriptFile : string * TextIO.instream -> unit (* Addded
by DAYA *)
val evalStream : TextIO.instream * Environment.environment ->
Environment.environment
```

b) New function definition for useScriptFile' is added to interact.sml. Function 'useScriptFile' takes in script file and stream as arguments and pass the same to EvalLoop.evalStream to process the stream over REPL. From the new structure 'Mutecompiler', functions 'silenceCompiler', 'unsilenceCompiler' and

‘printStashedCompilerPutput’ are called-in here to perform specific tasks which are part of the key requirements. We will discuss more in detail on these functions later in this document. In short, ‘silenceCompiler’ is used to save the compiler output to a reference cell, ‘unsilenceCompiler’ will revert back the saved compiler output from reference cell to compiler and ‘printStashedCompilerPutput’ will print the saved or stashed output to the terminal for user.

Before the file stream is passed to EvalLoop.evalStream for processing, the compiler messages are muted, a dummy function is called (which does nothing) and the compiler messages are unmuted. This task is very much needed to make the script output clear of unnecessary messages presented to the user. This will supress the auto-loading messages for structure ‘Mutecompiler’ in the script output which is totally unnecessary and could be avoided for pretty printing.

Source code:

```
    fun useScriptFile (fname, stream) = (
      Mutecompiler.silenceCompiler () ;
      EvalLoop.evalStream ("<instream>", (TextIO.openString
"Backend.Mutecompiler.mcdummyfn ();") ) ;
      Mutecompiler.unsilenceCompiler () ;

      (EvalLoop.evalStream (fname, stream))
        handle exn => (
          Mutecompiler.printStashedCompilerOutput ();
          Mutecompiler.unsilenceCompiler ();
          EvalLoop.uncaughtExnMessage exn
          )
      )
```

In case of error in the script file, the compiler will throw the error message along with file name and line number information. And also it prints the details of the exception thrown to the terminal. But this is not the case if the user has decided to mute the compiler by calling function ‘silenceCompiler’. When the compiler messages are muted, the error messages are stashed and unavailable to be printed to the terminal. These messages need to be retrieved from stash and printed to the user; and the compiler need to be unmuted to print the exception details. Function ‘printStashedCompilerPutput’ is called to print the stashed

output to user and function ‘unsilenceCompiler’ is called to unmute the compiler before throwing the uncaught exception, so that the exception details are printed to the terminal.

7.3.4 backend.sig

A new structure Mutecompiler is declared within signature BACKEND,

Source code:

```
signature BACKEND = sig
    structure Profile : PROFILE
    structure Compile : COMPILE
    structure Interact : INTERACT
    structure Mutecompiler : MUTECOMPILER
    structure Machine : MACHINE
    val architecture: string
    val abi_variant: string option
end
```

7.3.5 backend-fn.sml

New structure Mutecompiler is defined within functor BackendFn,

Source code:

```
structure Mutecompiler = Mutecompiler
```

7.3.6 mutecompiler.sig

New signature MUTECOMPILER is defined with all the global variables and functions that are part of Structure Mutecompiler. The details of these variables and functions are described in the next section.

Source code:

```
signature MUTECOMPILER =
  sig
    val printlineLimit : int ref
    val compilerMuted : bool ref
    val isNewline : char -> bool
    val push : 'a list ref -> 'a -> unit
```

```
    val installPrintingLimitSettings : int list -> unit
    val saveControlPrintOut : unit -> unit
    val stashCompilerOutput : string -> unit
    val savePrintingLimits : unit -> unit
    val lowerPrintingLimitsToMin : unit -> unit
    val restoreControlPrintOut : unit -> unit
    val restorePrintingLimits : unit -> unit
    val outputFlush : TextIO.outstream -> TextIO.vector -> unit
    val silenceCompiler : unit -> unit
    val unsilenceCompiler : unit -> unit
    val printStashedCompilerOutput : unit -> unit
    val mcdummyfn : unit -> unit
end  (* signature MUTECOMPILER *)
```

7.3.7 mutecompiler.sml

New structure Mutecompiler is defined with global variables, core functions and sub-functions which are required to perform the tasks like muting the compiler, unmuting the compiler, printing the stashed compiler messages back to the terminal and increase the print limits.

a. ‘silenceCompiler’ function is defined to mute the compiler messages. The compiler messages are supressed by saving all the ‘Control.Print.out’ values to a reference cell. It also saves the current printing limits in a ref cell and then set them all to zero.

   Source code:

```
   fun silenceCompiler () =
           (compilerMuted := true;
            saveControlPrintOut ();
            Control.Print.out := { flush = fn () => (), say
= stashCompilerOutput };
            savePrintingLimits ();
            lowerPrintingLimitsToMin ());

fun saveControlPrintOut () =
            if isSome (! savedControlPrintOut)
            then ()
            else savedControlPrintOut := SOME (!
Control.Print.out);
```

```
fun stashCompilerOutput string
      = case String.fields isNewline string
         of nil => raise (Fail "impossible ")
          | [chunk] => push compilerOutputCurrentLine chunk
          | chunk :: lines
            => (if chunk <> "" then push
compilerOutputCurrentLine chunk else ();
                push compilerOutputPreviousFullLines
                     (String.concat (rev (!
compilerOutputCurrentLine)));
                let val (last :: others) = rev lines
                in app (push
compilerOutputPreviousFullLines)
                       (rev others);
                   compilerOutputCurrentLine
                   := (if last <> "" then [last] else [])
                end);

  fun savePrintingLimits () =
            if isSome (! savedPrintingLimitSettings)
            then ()
            else savedPrintingLimitSettings := SOME (map !
printingLimitRefs);

fun lowerPrintingLimitsToMin () =
          List.app (fn r => r := 0) printingLimitRefs;
```

b. ‘unsilenceCompiler’ function unmutes the compiler messages by restoring the printing limits and value of ‘Control.Print.out’.

Source code:

```
   fun unsilenceCompiler () = (compilerMuted := false;
                               restoreControlPrintOut ();
                               restorePrintingLimits ());

       fun restoreControlPrintOut () =
               case ! savedControlPrintOut of
                   NONE => ()
                 | SOME value => (savedControlPrintOut :=
   NONE;
```

```
                                    Control.Print.out :=
    value);

        fun restorePrintingLimits () =
                case ! savedPrintingLimitSettings of
                    NONE => ()
                  | SOME settings =>
    (savedPrintingLimitSettings := NONE;

    installPrintingLimitSettings settings);
```

c. ‘printStashedCompilerOutput’ function is defined to print the saved or stashed compiler output back to the terminal. Whenever the compiler messages are supressed through ‘silenceCompiler’ function and an error occurs, these error messages are stashed and need to be printed back to the terminal. By invoking this function, the stashed messages can be printed back to the terminal.

d. ‘dummyfn’ function is created (which does nothing) and is called to preload the ‘Mutecompiler’ structure before the script is passed to EvalLoop.evalStream function. This is to supress the structure auto-loading logs in the script results. If this dummy function is not called, then the user can see the auto-loading logs in the script result when one of the ‘Mutecompiler’ function or variable is accessed. This is more of a cosmetic feature, to make sure the script results don’t have unnecessary logs in it.

e. Variable ‘printlineLimit’ is used to increase or decrease the number of lines are retrieved from the stashed output before they are printed to the terminal. By default, this limit is set to 5, but the user can change this limit if they want to see more details from the stashed output.

   The limit can be changed as below,

```
Backend.Mutecompiler.printlineLimit := 10;
```

7.3.8 INDEX, MAP and core.cm

INDEX, MAP and core.cm files are updated with definitions for signature 'MUTECOMPILER' and structure 'Mutecompiler', this will let CM know there is a new signature and structure defined and to be made available for usage.

a. In 'INDEX' file the definitions are added as below,

```
MUTECOMPILER
  TopLevel/interact/mutecompiler.sig
Mutecompiler
  TopLevel/interact/mutecompiler.sml
```

b. In 'MAP' file the definitions are added as below,

```
interact/
    envref.sml
      supports top-level environment management
      defs: ENVREF, EnvRef : ENVREF
    evalloop.sig,sml
      top-level read-eval-print loop
      defs: EVALLOOP, EvalLoopF: TOP_COMPILE => EVALLOOP
    interact.sig,sml
      creating top-level loops
      defs: INTERACT, Interact: EVALLOOP => INTERACT
    mutecompiler.sig,sml
      allow compiler silencing
      defs: MUTECOMPILER, Mutecompiler
```

c. In 'core.cm' file the definitions are added as below,

```
TopLevel/interact/mutecompiler.sig
TopLevel/interact/mutecompiler.sml
```

7.4 Writing a script

The script should start with '#!' in the first line followed by the environment location, command-line parameters '-Ssml' and '—script', a new line and then followed by the SML code or program.

Example script named 'sample',

```
--------------beginning of the script-------------
#!/usr/bin/env -Ssml -script
;(*-*-SML-*-*)
val () = print "Hello World\n";
--------------end of the script---------------------
```

7.5 Executing a script

The script ‘sample’ can be executed from Linux terminal or command prompt as a regular OS script as below (provided it is given execute permission),

```
$ ./sample
```

7.6 SML/NJ Implementation and 64-bit Support

As mentioned in section 7.1 the change was initially started with 32-bit version of SML/NJ 110.99.3. As part of our change, we are preloading structure Mutecompiler through a dummy function to avoid auto-loading logs in the output. This in-turn preloads structure Backend and solved the issue reported in 7.1. We were able to test our changes successfully in SML/NJ 110.99.3 64-bit version. There is no separate code base for 32-bit and 64-bit, the code base is same and only installation instruction for SML/NJ has a minor variation for 32-bit, which is as per the standard implementation instructions.

SML/NJ can be implemented in one of two methods described below,

1. Installation file config.tgz downloaded directly from SML/NJ portal (https://smlnj.org/dist/working/110.99.3/index.html) and follow the command line instructions in the same portal. The installation should include source file and re-compiling support, and this is achieved by enabling request for source in ‘targets’ file (part of installation files in ‘config’ directory).
2. Installation from github repository smlnj/legacy (https://github.com/smlnj/legacy). The installation and re-compile instructions are available in the same portal.

SML/NJ package installation is available which doesn’t have the support for source code and recompiling, it is recommended to follow one of two installation procedures recommended above.

7.7 GitHub Codebase

Codebase for this dissertation is forked from SML/NJ's github repository 'legacy - https://github.com/smlnj/legacy'. The code is forked into author's github repository (https://github.com/dn2007hw/legacy).

The changes are available in two branches as below,

1) Branch 'dn2007hw-patch-1-execute-as-a-script' include change for execute as a script functionality. (https://github.com/dn2007hw/legacy/tree/dn2007hw-patch-1-execute-as-a-script)
2) Branch 'dn2007hw-patch-2-with-mutecompiler' include changes for execute as a script capability and Mutecompiler functions. (https://github.com/dn2007hw/legacy/tree/dn2007hw-patch-2-with-mutecompiler)

Automated test suite is available in a separate branch 'ForFullTest'.

7.8 Merging with SML/NJ

Two separate pull requests have been raised with SML/NJ to merge the changes into SML/NJ legacy repository and they are under review by the SML/NJ implementors.

The pull requests are available as below,

1. #276 for 'Execute as a script' from branch 'dn2007hw-patch-1-execute-as-a-script', https://github.com/smlnj/legacy/pull/276.
2. #275 for 'Execute as a script with Mutecompiler' from branch 'dn2007hw-patch-2-with-mutecompiler', https://github.com/smlnj/legacy/pull/275.

Along the development of this change, we raised few issue requests with SML/NJ team in github to understand few concepts and behaviour, here are the link to those requests,

- https://github.com/smlnj/legacy/issues/268
- https://github.com/smlnj/legacy/issues/273
- https://github.com/smlnj/legacy/issues/277

8. Evaluation and Results

8.1 Before the change

As we saw in section 2.13, SML doesn't have the capability to allow a SML source file to be executed like a script. Let's see an example, a simple script 'sample' as below,

```
$ cat sample
#!/usr/bin/env -Ssml
;(* SML code starts here *)
val x = "Hello World x\n";
val () = print x;
$
```

When the above script is executed with the SML logs enabled, it failed with the below error messages,

```
$ ./sample
.....
exec "$RUN" @SMLcmdname="$0" "$HEAP" $ALLOC "$@"
+ exec /usr/local/smlnj/bin/.run/run.amd64-darwin
@SMLcmdname=/usr/local/smlnj/bin/sml @SMLload=./sample
/usr/local/smlnj/bin/sml: Fatal error -- incorrect byte order
in heap image
$
```

Though the program loader recognised the interpreter as SML, the SML was not able to load its heap image due to incorrect arguments passed on to the interpreter. If we add an additional argument ('--script') to the same script,

```
$ cat sample
#!/usr/bin/env -Ssml --script
;(* SML code starts here *)
val x = "Hello World x\n";
val () = print x;
$
```

The program loader recognised the interpreter as SML, the SML got loaded with heap image and the remaining arguments were passed on to the interpreter. Due to unavailability of the feature, SML was unable to recognise the argument '--script' and process the file; it continued to interactive mode.

```
$ ./sample
….
exec "$RUN" @SMLcmdname="$0" "$HEAP" $ALLOC "$@"
```

```
+ exec /usr/local/smlnj/bin/.run/run.amd64-darwin
@SMLcmdname=/usr/local/smlnj/bin/sml
@SMLload=/usr/local/smlnj/bin/.heap/sml --script ./sample
Standard ML of New Jersey (64-bit) v110.99.3 [built: Thu Jul 28
00:35:16 2022]
!* unable to process '--script' (unknown extension '<none>')
!* unable to process './sample' (unknown extension '<none>')
- ^D
$
```

8.2 After the change

8.2.1 Execute as a script.

Post implementation of the changes, we were able to execute SML source code as scripts. Let's consider the same example,

```
$ cat sample
#!/usr/bin/env -Ssml --script
;(* SML code starts here *)
val x = "Hello World x\n";
val () = print x;
$
```

Now that we have implemented the capability, we were able to execute the source code as a script,

```
$ ./sample
Standard ML of New Jersey (32-bit) v110.99.3 [built: Sun Apr 23
20:39:28 2023]
val x = "Hello World x\n" : string
Hello World x
$
```

By enabling the logs to verify the local variables, we can see the program loader able to identify the SML interpreter and pass on the arguments to the interpreter.

```
$ ./sample
…
…
exec "$RUN" @SMLcmdname="$0" "$HEAP" $ALLOC "$@"
+ exec /Users/ndaya/withmc32/bin/.run/run.x86-darwin
@SMLcmdname=/Users/ndaya/withmc32/bin/sml
@SMLload=/Users/ndaya/withmc32/bin/.heap/sml --script ./sample
Standard ML of New Jersey (32-bit) v110.99.3 [built: Sun Apr 23
20:39:28 2023]
val x = "Hello World x\n" : string
Hello World x
$
```

Below is the log when SML is loaded directly from the command prompt and the arguments passed to the interpreter.

```
exec "$RUN" @SMLcmdname="$0" "$HEAP" $ALLOC "$@"
+ exec /Users/ndaya/withmc32/bin/.run/run.x86-darwin
@SMLcmdname=/Users/ndaya/withmc32/bin/sml
@SMLload=/Users/ndaya/withmc32/bin/.heap/sml
Standard ML of New Jersey (32-bit) v110.99.3 [built: Sun Apr 23
20:39:28 2023]
```

Below is the log when a SML source code (‘sample’) is executed as a script, we can see the interpreter is loaded with default parameters @SMLcmdname and @SMLload, followed by newly introduced argument ‘--script’ and the script file name ‘sample’.

```
exec "$RUN" @SMLcmdname="$0" "$HEAP" $ALLOC "$@"
+ exec /Users/ndaya/withmc32/bin/.run/run.x86-darwin
@SMLcmdname=/Users/ndaya/withmc32/bin/sml
@SMLload=/Users/ndaya/withmc32/bin/.heap/sml --script ./sample
Standard ML of New Jersey (32-bit) v110.99.3 [built: Sun Apr 23
20:39:28 2023]
```

The newly introduced changes as described in section 7.3.1, 7.3.2 and 7.3.3 have recognised the script, loaded the SML and executed the instructions in the script file.

8.2.2 Silencing compiler messages

We were also able to successfully verify the capability we have introduced to silence the compiler messages. This is achieved by introducing a new structure Mutecompiler and custom built functions within it. The features include, silencing/muting compiler messages, un-silencing/unmuting compiler messages, printing silenced messages and increase the print limits while printing silenced messages.

Let's consider the same example,

```
$ cat sample
#!/usr/bin/env -Ssml --script
;(* SML code starts here *)
val x = "Hello World x\n";
val () = print x;
$ ./sample
Standard ML of New Jersey (32-bit) v110.99.3 [built: Sun Apr 23
20:39:28 2023]
val x = "Hello World x\n" : string
Hello World x
$
```

In the above result, the 'val x = "Hello World x\n" : string' is the compiler message printed as part of the variable declaration. And 'Hello World x' is the output of the print instruction. One of the goals for our dissertation is to introduce the capability to hide/silence/mute these compiler messages. The changes we have introduced as described in sections 7.3.3 thru 7.3.8 have enabled this capability. By calling the appropriate functions from structure Mutecompiler we can achieve the desired results.

Let's consider the same example, now call the silenceCompiler function to mute compiler messages,

```
$ cat sample
#!/usr/bin/env -Ssml --script
val _ = Backend.Mutecompiler.silenceCompiler ();
val x = "Hello World x\n";
val () = print x;
```

```
$ ./sample
Standard ML of New Jersey (32-bit) v110.99.3 [built: Sun Apr 23
20:39:28 2023]
Hello World x
$
```

We can see the compiler message is muted but not the output. The muting takes effect only for the compiler messages that are generated from the point where the function is called.

Let's consider another example and result,

```
$ cat sample
#!/usr/bin/env -Ssml --script
val x = "Hello World x\n";
val () = print x;
val _ = Backend.Mutecompiler.silenceCompiler ();
val y = "Hello World y\n";
val () = print y;
$ ./sample
Standard ML of New Jersey (32-bit) v110.99.3 [built: Sun Apr 23
20:39:28 2023]
val x = "Hello World x\n" : string
Hello World x
Hello World y
$
```

In the above results we can see the compiler message generated as part of variable declaration before invoking the silenceCompiler function is been printed to the output. The compiler message for the variable declaration after invoking the silenceCompiler function has been supressed or muted.

8.2.3 Un-silencing compiler messages

We were able to verify the capability to un-mute the compiler messages by invoking the function unsilenceCompiler. This is only required if the compiler messages are muted earlier. This feature gives us the capability to supress the compiler messages for a selective part of the source code.

Let’s consider the example and result,

```
$ cat sample
#!/usr/bin/env -Ssml --script
val x = "Hello World x\n";
val () = print x;
val _ = Backend.Mutecompiler.silenceCompiler ();
val y = "Hello World y\n";
val () = print y;
val _ = Backend.Mutecompiler.unsilenceCompiler ();
val z = "Hello World z\n";
val () = print z;
$ ./sample
Standard ML of New Jersey (32-bit) v110.99.3 [built: Sun Apr 23
20:39:28 2023]
val x = "Hello World x\n" : string
Hello World x
Hello World y
val z = "Hello World x\n" : string
Hello World z
$
```

From the results above we can see the compiler messages are suppressed for declaration of variable ‘y’. The calling of function silenceCompiler before variable declaration and then calling function unsilenceCompiler after the variable declaration for ‘y’ have supressed the compiler the messages. The compiler messages for declarations before and after the functions were printed.

8.2.4 Printing silenced compiler messages

Silencing the compiler message helps to generate a pretty out but it also supresses the error messages. In case there is an error in the instructions passed from script file, the details of the error won’t be printed out to the user.

Let’s consider the example with a forced error in the variable declaration,

```
$ cat sample
#!/usr/bin/env -Ssml --script
```

```
val x == "Hello World x\n";
val () = print x;
$ ./sample
Standard ML of New Jersey (32-bit) v110.99.3 [built: Sun Apr 23
20:39:28 2023]
./sample:2.27-3.4 Error: syntax error: deleting  SEMICOLON VAL

uncaught exception Compile [Compile: "syntax error"]
  raised at: ../compiler/Parse/main/smlfile.sml:19.24-19.46
             ../compiler/TopLevel/interact/evalloop.sml:45.54
             ../compiler/TopLevel/interact/evalloop.sml:306.20-
306.23
$
```

In the above example, we can see the error details called out with the file name (./sample) and specific line number (./sample:2.27-3.4) in the file where the error have occurred.

Let’s consider an example where the compiler messages are silenced and an error occurs,

```
$ cat sample
#!/usr/bin/env -Ssml --script
val x = "Hello World x\n";
val () = print x;
val _ = Backend.Mutecompiler.silenceCompiler ();
val y == "Hello World y\n";
val () = print y;
$ ./sample
Standard ML of New Jersey (32-bit) v110.99.3 [built: Sun Apr 23
20:39:28 2023]
val x = "Hello World x\n" : string
Hello World x

uncaught exception Compile [Compile: "syntax error"]
  raised at: ../compiler/Parse/main/smlfile.sml:19.24-19.46
             ../compiler/TopLevel/interact/evalloop.sml:45.54
             ../compiler/TopLevel/interact/evalloop.sml:306.20-
306.23
$
```

In the above result we can see the output doesn't have any info on what the error is and where the error is occurring, this is because we have silenced the compiler messages. This is where the new function 'printStashedCompilerOutput' comes in to get these error messages printed. The users don't need to call the function or invoke in scripts, this is handled automatically. The above example is only for illustration.

Let's consider an example where the compiler messages are silenced and an error occurs, now the function 'printStashedCompilerOutput' is implemented.

```
$ cat sample
#!/usr/bin/env -Ssml --script
val x = "Hello World x\n";
val () = print x;
val _ = Backend.Mutecompiler.silenceCompiler ();
val y == "Hello World y\n";
val () = print y;
$ ./sample
Standard ML of New Jersey (32-bit) v110.99.3 [built: Sun Apr 23
20:39:28 2023]
val x = "Hello World x\n" : string
Hello World x
_____________________________________________________________
____
The last 5 lines 30 through 34 of suppressed compiler messages
are:
[library $smlnj/MLRISC/IA32.cm is stable]
[library $SMLNJ-MLRISC/IA32.cm is stable]
[autoloading done]
val it = # : unit
./sample:5.27-6.4 Error: syntax error: deleting  SEMICOLON VAL
_____________End of suppressed compiler
messages._____________________

uncaught exception Compile [Compile: "syntax error"]
  raised at: ../compiler/Parse/main/smlfile.sml:19.24-19.46
             ../compiler/TopLevel/interact/evalloop.sml:45.54
             ../compiler/TopLevel/interact/evalloop.sml:306.20-
306.23
```

```
$
```

In the above result we can see the error messages printed to the output. The compiler messages that got stashed are retrieved and printed to the output by the new function. By default, only the last 5 lines of the stashed messages are retrieved and printed. This line limit can be increased or decreased based on user needs.

Let’s consider this example with an error, compiler messages silenced and print line limit increased to 10.

```
$ cat sample
#!/usr/bin/env -Ssml --script
Backend.Mutecompiler.printlineLimit := 10;
val x = "Hello World x\n";
val () = print x;
val _ = Backend.Mutecompiler.silenceCompiler ();
val y = "Hello World y\n";
val () = print 3;
$ ./sample
Standard ML of New Jersey (32-bit) v110.99.3 [built: Sun Apr 23
20:39:28 2023]
val x = "Hello World x\n" : string
Hello World x
_____________________________________________________________
____
The last 10 lines 30 through 39 of suppressed compiler messages
are:
[library $smlnj/MLRISC/IA32.cm is stable]
[library $SMLNJ-MLRISC/IA32.cm is stable]
[autoloading done]
val it = # : unit
val y = # : string
./sample:7.5-7.17 Error: operator and operand do not agree
[overload - bad instantiation]
  operator domain: string
  operand:         'Z[INT]
  in expression:
    <exp>
```

```
_____________End of suppressed compiler
messages._____________________

uncaught exception Error
  raised at: ../compiler/TopLevel/interact/evalloop.sml:69.13-
69.21
             ../compiler/TopLevel/interact/evalloop.sml:45.54
             ../compiler/TopLevel/interact/evalloop.sml:306.20-
306.23$
$
```

From the result above, we can see the last 10 lines of the stashed compiler messages are printed.

8.2.5 Restoring printing limits

From the examples in the previous section, you may notice the compiler message for variable declarations will differ from the compiler messages that are stashed. This is because we have reset the printing limits to avoid saving all the variable information in the memory.

```
val x = "Hello World x\n" : string
(vs)
val y = # : string
```

This information can be restored by pre-setting the printing limits. Lets see the sample example with printing limits restored.

```
$ cat sample
#!/usr/bin/env -Ssml --script
Backend.Mutecompiler.printlineLimit := 10;
val x = "Hello World x\n";
val () = print x;
val _ = Backend.Mutecompiler.silenceCompiler ();
val _ = Backend.Mutecompiler.restorePrintingLimits ();
val y = "Hello World y\n";
val () = print 3;
$ ./sample
Standard ML of New Jersey (32-bit) v110.99.3 [built: Sun Apr 23
20:39:28 2023]
```

```
val x = "Hello World x\n" : string
Hello World x
_____________________________________________________________________
____
The last 10 lines 30 through 39 of suppressed compiler messages
are:
[library $smlnj/MLRISC/IA32.cm is stable]
[library $SMLNJ-MLRISC/IA32.cm is stable]
[autoloading done]
val it = # : unit
val y = "Hello World y\n" : string
./sample:7.5-7.17 Error: operator and operand do not agree
[overload - bad instantiation]
  operator domain: string
  operand:         'Z[INT]
  in expression:
    <exp>
_____________End of suppressed compiler
messages._____________________

uncaught exception Error
  raised at: ../compiler/TopLevel/interact/evalloop.sml:69.13-
69.21
             ../compiler/TopLevel/interact/evalloop.sml:45.54
             ../compiler/TopLevel/interact/evalloop.sml:306.20-
306.23$
$
```

In the results above you can see the compiler message for variable declaration of ‘y’ saved in stash without any change.

8.3 Sample Scripts

Here are some sample scripts and their results.

8.3.1 Example #1:

Script:

```
$ cat sample1
#!/usr/bin/env -Ssml --script
;(*-*-SML-*-*)
fun cube x = x * x * x ;
```

```
fun square x = x * x ;
val () = print (Int.toString(square(9)) );
val () = print "\n";
val () = print (Int.toString(cube(3)) );
val () = print "\n";
print (Int.toString(cube(10)) );
$
```

Result:

```
$ ./sample1
Standard ML of New Jersey (32-bit) v110.99.3 [built: Sun
Apr 23 20:39:28 2023]
val cube = fn : int -> int
val square = fn : int -> int
[autoloading]
[autoloading done]
81
27
1000val it = () : unit
$
```

8.3.2 Example #2:

Script:

```
$ cat sample2
#!/usr/bin/env -Ssml --script
;(*-*-SML-*-*)
val _ = Backend.Mutecompiler.silenceCompiler ();
fun cube x = x * x * x ;
fun square x = x * x ;
val () = print (Int.toString(square(9)) );
val () = print "\n";
val () = print (Int.toString(cube(3)) );
val () = print "\n";
print (Int.toString(cube(10)) );
$
```

Result:

```
$ ./sample2
Standard ML of New Jersey (32-bit) v110.99.3 [built: Sun
Apr 23 20:39:28 2023]
81
27
```

1000$

9. Conclusion and Future Work

9.1 Conclusion

This dissertation intended to make SML programming simpler for developers, programmers, and students such that they can write SML source code in a file and execute it as a script. And to integrate *smlnj-script* capabilities into SML/NJ and do away with the requirement to maintain a separate *smlnj-script* implementation.

By analysing the SML/NJ installation procedures, scripts, and logs, we were able to understand how the SML/NJ is built on both C and SML language codebase. Every time SML is loaded, the C executables load the SML heap image which is built as part of the installation process and takes in command line arguments to perform desired functions. This was the entry point of our dissertation. We intercepted the modules which received the command line arguments and identified the right place to add our changes so that the script execution will be recognised, and the instructions are passed through evaluation loop.

We were able to implement one of the key *smlnj-script* capability, silencing the compiler, which is to supress the compiler output messages. We introduced a new structure Mutecompiler with built-in functions which gave the capability to silence and un-silence compiler messages, and to restore the supressed messages for printing.

The results discussed in section 8 is a proof that this dissertation achieved the core objective and a high-level requirement.

9.2 Future Work

Though we implemented the main objective and couple of key features in this dissertation, there are few other nice to have features available in *smlnj-script* (described in section 2.8) that can be merged on to SML/NJ. It needs a considerate amount of time to analyse the SML/NJ code base, for example string overloading is like complete overhaul of string handling and due diligence is needed to analyse every usage of it in SML. The same goes to other features in *smlnj-script*. With the support and guidance from subject matter experts, these features appropriate modules will be identified, changes will be built and merged back into SML/NJ.

Bibliography

1. Wikipedia. (2022). *ML (programming language)*. [online] Available at: https://en.wikipedia.org/wiki/ML_(programming_language).
2. en.bmstu.wiki. (n.d.). *ML (Meta Language) - Bauman National Library*. [online] Available at: https://en.bmstu.wiki/ML_(Meta_Language) [Accessed 1 Dec. 2022].
3. Wikipedia. (2022). *Standard ML*. [online] Available at: https://en.wikipedia.org/wiki/Standard_ML [Accessed 1 Dec. 2022].
4. www.smlnj.org. (n.d.). *Standard ML*. [online] Available at: http://www.smlnj.org/sml.html [Accessed 1 Dec. 2022].
5. R Milner (1997). *The definition of standard ML : revised*. Cambridge, Mass.: Mit Press.
6. www.smlnj.org. (n.d.). *SML '97*. [online] Available at: http://www.smlnj.org/sml97.html [Accessed 1 Dec. 2022].
7. J Wells, (2012). *The Standard ML Programming Language*. [online] www.macs.hw.ac.uk. Available at: http://www.macs.hw.ac.uk/~jbw/sml.html.
8. www.smlnj.org. (n.d.). *SML/NJ background information*. [online] Available at: https://smlnj.org/smlnj.html [Accessed 1 Dec. 2022].
9. Wikipedia. (2022). *Standard ML of New Jersey*. [online] Available at: https://en.wikipedia.org/wiki/Standard_ML_of_New_Jersey [Accessed 1 Dec. 2022].
10. J Wells, *SML/NJ Scripts and the "smlnj-script" Interpreter*. [online] Available at: https://github.com/ultra-group/smlnj-script.
11. Robin Milner. A theory of type polymorphism in programming. Journal of Computer and System Sciences, 17(3):348–375, 1978.
12. Robin Milner; Mads Tofte; Robert Harper; David MacQueen (1997). The Definition of Standard ML (Revised). MIT Press. ISBN 0-262-63181-4.
13. www.smlnj.org. (n.d.). *Standard ML of New Jersey License*. [online] Available at: https://www.smlnj.org/license.html [Accessed 1 Dec. 2022].
14. linux.die.net. (n.d.). *execve(2): execute program - Linux man page*. [online] Available at: https://linux.die.net/man/2/execve.
15. www.ibm.com. (n.d.). *Compiled versus interpreted languages*. [online] Available at: https://www.ibm.com/docs/en/zos-basic-skills?topic=zos-compiled-versus-interpreted-languages.

16. Hawaii.edu. (2019). [online] Available at: http://www2.hawaii.edu/~takebaya/ics111/process_of_programming/process_of_programming.html.
17. hjaem.info. (n.d.). *First-Class Continuations*. [online] Available at: https://hjaem.info/articles/en_18_4#:~:text=First%2Dclass%20continuations%20are%20continuations [Accessed 1 Dec. 2022].
18. www.smlnj.org. (n.d.). *SML/NJ Special Features*. [online] Available at: https://www.smlnj.org/doc/features.html [Accessed 1 Dec. 2022].
19. Wikipedia. (2022). *Functor (functional programming)*. [online] Available at: https://en.wikipedia.org/wiki/Functor_(functional_programming) [Accessed 1 Dec. 2022].
20. teaching.idallen.com. (n.d.). *Shell Command Line Quoting Mechanisms*. [online] Available at: https://teaching.idallen.com/cst8207/13w/notes/440_quotes.html#why-do-we-need-a-shell-quoting-mechanism [Accessed 1 Dec. 2022].
21. www.classes.cs.uchicago.edu. (n.d.). *Basic Compiling*. [online] Available at: https://www.classes.cs.uchicago.edu/archive/2009/fall/51081-1/LabFAQ/introlab/compile.html [Accessed 1 Dec. 2022].
22. www.cs.cmu.edu. (n.d.). *Using the SML/NJ System*. [online] Available at: https://www.cs.cmu.edu/afs/cs/local/sml/common/smlguide/smlnj.htm [Accessed 1 Dec. 2022].
23. R Lakshmanan, *'What is Garbage collection log, Thread dump, Heap dump?'* [online] Available at: https://devm.io/java/heap-dump-gc-173352#:~:text=Heap%20dumps%20are%20primarily%20used%20for%20troubleshooting%20memory%20related%2C%20OutOfMemoryError%20problems [Accessed 1 Dec. 2022].
24. M. Blume, 2002, 'CM The SML/NJ Compilation and Library Manager' [online] Available at: https://www.smlnj.org/doc/CM/new.pdf
25. D. MacQueen, 2007, www.smlnj.org. *heap2exec*. [online] Available at: https://www.smlnj.org/doc/heap2exec/index.html [Accessed 6 Dec. 2022].
26. Wikipedia. (2023). *Shebang (Unix)*. [online] Available at: https://en.wikipedia.org/wiki/Shebang_(Unix) [Accessed 24 Apr. 2023].
27. fxp - a Functional XML Parser (no date). Available at: http://www2.cs.tum.edu/projects/Fxp/.http://www2.cs.tum.edu/projects/Fxp/