TMSL (TURING MACHINE SIMULATION LANGUAGE)
Language Manual and Project Report

Isaac McGarvey (iam2108)
Joshua Gordon (jbg2109)
Keerti Joshi (kj2217)
Snehit Prabhu (sap2131)

Table of Contents

1. Introduction

1.1.1. Overview

A Turing Machine is a simple but powerful computer. It is useful in thinking about the nature and limits of computability because its method of computation is about as simple as can be imagined. Important theoretical results about what can be computed that are expressed in the terms of Turing Machines are clearer to intuition than the same results expressed in other terms.
The Turing Machine is an automaton whose temporary storage is a tape. This tape is divided into cells each of which is capable of holding one symbol. Associated with the tape is a read-write head that can travel left of write on the tape and that can read or write a single symbol on each move. The automaton that we use as a Turing Machine will have neither an input file nor any special output mechanism. Whatever input and output is necessary will be done on the machine's tape.
The Turing Machine Simulation Language (TMSL) is a programming language that is designed allow users to write programs that will be compiled into a Turing Machine that can execute the program. Because Turing Machines are very different from normal computers and also far more restrictive in terms of the number of available operations, TMSL is very different from a typical imperative programming language. Programming a Turing Machine by specifying states and transitions is analogous to programming a modern computer in assembly, at best. We hope that TMSL will be a high-level language (relatively speaking) for Turing Machines.

1.1.2. Example

As an example program we begin with the following which demonstrates unsigned addition. Comments and bracketing are c-style. Notice that the alphabet is specified on the first line of the program. This is mandatory, and serves the purpose of describing those symbols which may be written to and read from the tape. The empty symbol "_" is included by default.

```
/* alphabet specification */
0,1

/* true while the symbol under the tape head is 1 */
while (1){
   /* move the tape head one position to the right */
   right
   /* if the symbol under the tape head is 0 */
   if (0){
      /* move the tape head one position to the right */
      right

      /* if the symbol under the tape head is 1 */
      if (1){
         write 0 /* replace the current symbol under the tape head with 0 */
         left /* move the tape head one position to the left */
         write 1 /* replace the current symbol under the tape head with 1 */
      }

   }
}
```

2. Language Tutorial

This chapter represents a tutorial for a novice to get started with TMSL.

2.1 Compilation and Execution

A TMSL program consists initially of a plain text ".tmsl" file, which contains the user defined program. For now, let us consider the example program of 1.1.3, unsigned addition. Let us call this file "unsignedAdd.tmsl". While this file alone is a complete specification of a TMSL program, typically the user must also specify 1) the input tape for the Turing machine which is to execute the program, and 2) the output tape file, which is a blank target file overwriten with the final state of the tape when a Turing machine executes its program. As a convention, the input and output tape are given the extensions ".tape". The input tape is a plain text file of format: "L "*, where L is a symbol in the alphabet specified in the user program, followed by a space, repeated zero or more times. For instance, to add the numbers four and two, we specify an input.tape file as "1 1 1 1 0 1 1" (the contents of the input.tape are program specific). For the output tape, we specify a blank file, "output.tape".

Compilation: given "unsignedAdd.tmsl", we compile the program by: ".\compiler\compiler.exe unsignedAdd.tmsl transition.tm", where unsignedAdd.tmsl is the user specified program, and "transition.tm" is the target output file to contain the resulting compiled state transition table.

Execution: given "transition.tm", "input.tape", and "output.tape", a user may execute the program on a turning machine by ".\machine\machine.exe transition.tm input.tape output.tape". The output.tape file will be transformed into a plain text file containing the final state of the tape.

For convenience, we have included "run.bat" in the root TMSL directory. Run is a small batch file which automates the steps necessary to compile and execute user defined TMSL programs. To compile, execute, and display the final state of the tape, the user may command "run.bat unsignedAdd.tmsl input.tape output.tape"

3. Language Manual

3.1 Lexical Conventions

3.1.1 Tokens

There are three types of tokens in this language: identifiers (symbols), keywords,and separators. Tokens are whitespace delineated.

3.1.2 Comments

TMSL supports multi-line comments. A section of text beginning with **/*** and ending with ***/** is a comment. An example comment is as follows:

/* this is a comment */
/* this is another comment */

3.1.3 Symbols

Symbols are values / identifiers that can be read from and written to the tape.
A symbol is represented by a sequence of letters and digits. Letters are not case sensitive.

<symbol> = ['a'-'z' 'A'-'Z' '0'-'9']+

The only only special symbol is the symbol / keyword *'_' (to represent a blank*). All uninitialized cells of the tape contain the blank symbol. As with any other symbol, the '_' symbol can also be read from or written to the tape.

The set of all symbols a machine can read and write, excluding the '_' symbol, is called its alphabet.

3.1.4 Symbol Lists

A symbol list is a sequence of symbols separated by commas:

<comma> = ','

<symbol list> = (<symbol> <comma>)* <symbol>

3.1.5 Keywords

The following are reserved for use as keywords and may not be used as identifiers:

if
else
unless
while
until
left
right
write
exit
‘_’

3.1.6 Separators

The following are used to indicate separators:

,
{ }

3.1.7 Datatypes

The only data type in TMSL is that of the set of symbols that the machine can process. Each of the implicit variables in TMSL (that is, the cells on the tape) are of this data type by definition.

3.1.8 Variables

TMSL does not have any explicit variables. The cells of the tape act as variables, but only the cell that is currently underneath the head can be accessed. All reading and writing of the tape cells implicitly access the current cell of the tape, as in the condition part of the control statements and in the write instruction.

3.2 Declarations

The alphabet of the program (the set of symbols that will be used) must be declared in the symbol list at the beginning of the program. Though the blank symbol ('_') can be used in the program, it is not technically considered a member of the alphabet in our formal definition of a Turing machine, so it need not be listed in the symbol declaration.

3.3 Statements

A statement in TMSL can be one of the following types: if, ifelse, while, write, left, right, exit, unless, until

<statement> = <if-statement> | <ifelse-statement> | <unless-statement> | <while-statement> | <until-statement> | <write-statement> | <left-statement> | <right-statement> | <exit-statement> | <unless-statement> | <block-statement>

3.3.1 If

An if statement in TMSL usually looks like this:

if (<*symbol list*>)
{
   <*statement list*>
}

This means that if the symbol in the current cell of the tape is contained in the symbol list, the statement list is executed. Otherwise, the statement list is not executed.

The body of the if statement is technically a single statement, but that statement can be a block statement which itself contains multiple statements. So in practice, the braces of the around the body of the if statement are optional if the body contains only a single statement. This same rule also applies to the bodies of all the other control flow statements (if-else, unless, while, until).

<if-statement> = 'if' '(' <symbol list> ')' <statement>

3.3.1 IfElse

An ifelse is **NOT** sematically equivalent to the elseif construct of popular programming languages like PHP. An ifelse statement in TMSL looks like this:

if (<*symbol list*>)
{
   <*statement list*>
}
else
{
   <*statement list*>
}

This means that if the symbol in the current cell of the tape is contained in the symbol list, the statement list in the if block executed. Otherwise, the statement list in the else block is executed. Depending on the alphabet in the current cell of the tape, One of the two statement lists is always executed, since the symbol-list of the else block is the complement of the symbol list of if (i.e. {symbol-list of if} U {symbol-list of else} = Tape-Alphabet}

<ifelse-statement> = 'if' '(' <symbol list> ')' <statement> 'else' <statement>

3.3.1 Unless

An unless statement in TMSL looks like this:

unless (<*symbol list*>)
{
    <*statement list*>
}

This means that if the symbol in the current cell of the tape is **not** contained in the symbol list, the statement list is executed.

<unless-statement> = 'unless' '(' <symbol list> ')' <statement>

3.3.2 While

A while statement looks like this:
while (<*symbol list*>)
{
    <*statement list*>
}

This means that the statement list is executed repeatedly, as long as the symbol at the current cell is in the symbol list.

<while-statement> = 'while' '(' <symbol list> ')' <statement>

3.3.2 Until

A until statement looks like this:
until (<*symbol list*>)
{
    <*statement list*>
}

This means that the statement list is executed repeatedly, as long as the symbol at the current cell is **not** in the symbol list.

<until-statement> = 'until' '(' <symbol list> ')' <statement>

3.3.3 Write

A write statement looks like this:
write <*symbol*>

This writes the the symbol onto the current cell of the tape.

<write-statement> = 'write' <symbol>

3.3.4 Left

A left statement is simply the left keyword itself:
left

This statement moves the tape one cell to the left.

<left-statement> = 'left'

3.3.5 Right

A right statement is simply the right keyword itself:
right

This statement moves the tape one cell to the right.

<right-statement> = 'right'

3.3.6 Exit

An exit statement is simply the exit keyword itself:
exit

This statement halts the program execution immediately.

3.3.7 Block

A block statement is list of statements grouped with braces:

```
{
   <statement list>
}
```

Executing the block statement means executing the statement list.

<block-statment> = '{' <statment list> '}'

3.3.8 Statement Lists

A statement list is simply a sequence of statements. There is no separator necessary between the statements.

<statement list> = <statement>+

Semantically, each statement in a statement list is executed in order.

3.3.9 Overall program structure

A program in TMSL is simply a symbol list (the alphabet declaration) followed by a list of statements, which are executed in order.

<program> = <symbol list> <statement list>

3.4 Scope

Since there are no explicit variables in TMSL, there are no scope rules. The only thing that might be considered to have a scope are the symbols in the machine's alphabet. The scope of the alphabet symbols are global.

3.5 Functions

There are no built-in or user-specified functions in TMSL.

4. Project Plan

This section outlines the project planning aspects of TMSL: organization of the team, environment used and further planning aspects.

4.1 Project Process

Project planning was done within the framework of the deadlines. We tried to even the work out over the semester to avoid doing all the work right
at the end, and on the whole this was successful.

Weekly Meetings

Our group had regular meetings each Wednesday to plan the week ahead.
Specification of the language was done through a series of workshops which resulted initially in a fairly reasonable example (the example, submitted in the language reference manual hand in). From this, we formally specified the language grammar.
Specification of aspects like the static semantic analysis was done with unit tests, so that by the end of the project we had a suite of tests for what was and was not allowed on a semantic level in our language. Development was done in two parts while one half of the project involved the compiler the other involved developing a machine which could translate the states generated by the compiler to a turing machine.
At the end phase of the project, we joined these two together. People had their own areas of specialization and owned parts of the project.
Our project put a strong emphasis on testing, but it was for most part the responsibility of each developer to test their own code.

To keep every member updated major communication tool used was Gmail. We had all our documents in google docs so that each member has access to view and edit all the project related documents.

Subversion

We used Google code for subversion.
Our project repository can be accessed through the link below.
http://code.google.com/p/turing-machine-language/

4.3 Project Timeline

Shown in the table below are some of the goals and milestones achieved for
the project

1) Language Proposal - 24 Sep 2008

2) Language Design - 17 Oct 2008

3) Language Reference Manual - 22 Oct 2008

4) Implementation - The code base was incrementally built over the weeks, while simultaneously maintaining a regression test suite. Since the language is somewhat limited in scope, we built the basic functionality by End November (as per initially proposed LRM), and added additional functionality and keywords thence. The test suite ensured backward compatibility.

4.4 Roles and Responsibilities

Isaac - Machine simulator, general framework for the code generation, code generation for many of the statement types, regression test script for the end-to-end test cases

Josh - Parser, Scanner, AST, compiler test cases, various compiler statements, language design, initial idea

Keerti -Parser, scanner, ast, test cases, parts of code generation, Project management and Documentation,

Snehit - Language Design, Language to State transition logic, Parts of scanner, parser, some statement types in compiler, example test-case programs, Documentation.

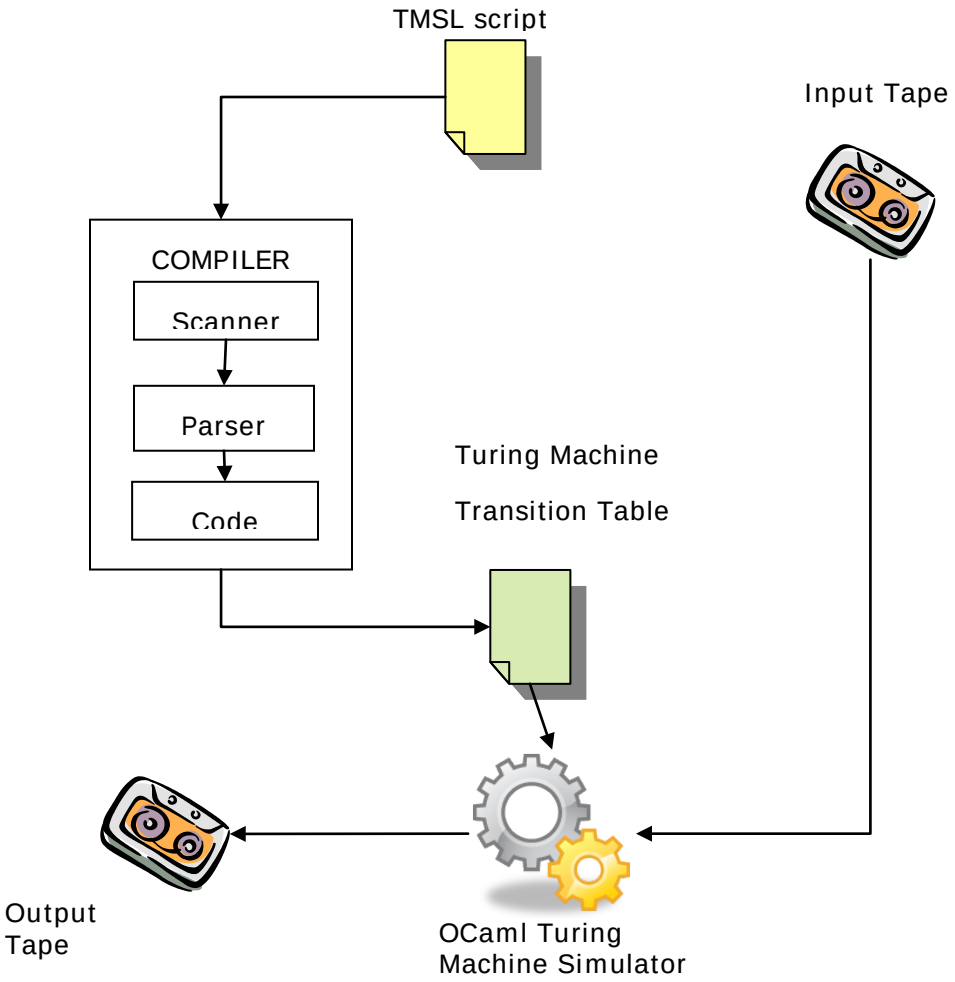
4.5 Development Environment

None. Notepad and basic text editors were enough. We did utilize Versioning systems (Google Code and Tortoise SVN) as well as other collaborative tools (Google docs).

4.6 Project Log

We didn't keep a project log, but the revision history for our repository can be found here:
http://code.google.com/p/turing-machine-language/source/list

5. Architecture



5.2 Description of Architecture

At the highest level of the architecture, our project consists of two main pieces: the compiler, and the Turing machine simulator. Each of those is a separate program. The compiler's job is to read the program as input and produce a description of the compiled Turing machine as output. The Turing machine simulator takes that description as input, along with a file containing the initial contents of the tape, and executes the machine, producing the final contents of the tape as output.

The interface between the compiler and the simulator is the file describing the Turing machine. The format of that file is simple. It contains a list of all the transitions in the machine. The transitions are separated by newlines. Each transition contains five pieces of information:
1. The current state of the tape.
2. The symbol that was read from the tape.

3. The next state of the tape.
4. The symbol to write to the tape.
5. How to move the head (left, right, stay).

Those items are separated by whitespace. Examples of this file can be found in section 6.4.

Note:
The machine simulator is hard-coded to start in the state called **s0**, but the other states can be given any alphanumeric name. Also, the machine execution stops when it can find no appropriate transition to follow.

Drilling down inside the top-level modules, the compiler module is composed of three main pieces: scanner, parser, and code generation. The scanner turns the text of the input program into a stream of tokens. The parser reads the tokens from the parser, checks that the program is valid according to the grammar of the language, and outputs the AST. The code generation part of the program does a recursive walk of the AST. It checks that all symbols used in the program have been declared and writes the state transitions for each node of the AST.

The machine simulator module has a scanner and parser that reads the Turing machine definition produced by the compiler. The output of that parser is simply a list of transitions. The machine simulator also has another scanner and parser that reads in the input tape file. The output of that parser is a list of symbols. Using scanners and parsers for reading in these simply-formatted input files may not have been necessary, but it seemed like an easy way to read in files in O'Caml. The main part of the machine simulator just follows the transitions in the transition list and updates the tape and the state of the machine accordingly after each transition it follows.

6. Testing

We used separate regression test suites for the compiler (converting the scripting user language into TM states) and the Turing Machine Engine (executing the state transitions based on user input tape content). A third end-to-end regression test suite was used to test the aggregate control flow through the whole pipeline.

6.1 Test Suites 1 - Compiler (./compiler/test)

The tests and the regression script in this directory were used for initial testing of the scanner and parser before the code generation and end-to-end regression script were written. The regression script (./compiler/regressionTest.bat) was written by Josh. It compiles each test so that syntax and parsing errors can be seen.

6.2 Test Suite 2 - Machine simulator (./machine/test)

There are a couple test files in this directory that were used in testing the functionality of the machine simulator. No automated regression testing was required here because the machine simulator was the first piece we implemented and only minor changes were made to it after that.

6.3 Test Suite 3 - End-to-end (./test)

The end-to-end regression script (./test/regression.ps1) was written by Isaac in PowerShell. It does the following for each of its test programs: compiles the program, runs the compiled machine on a certain input, and checks that the output matches the expected output. All team members contributed to writing the test cases.

6.4 Examples

Below is an example of one of our simplest test files (./test/if.tml). It is intended to merely test the basic functionality of the if statement.

```
0,1,2

if (0) write 1
if (0) write 2
```

This is the compiled machine:

```
s1 _ s2 1 stay
s1 2 s2 1 stay
s1 1 s2 1 stay
s1 0 s2 1 stay
s0 _ s2 _ stay
s0 2 s2 2 stay
s0 1 s2 1 stay
s0 0 s1 0 stay
```

```
s3 _ s4 2 stay
s3 2 s4 2 stay
s3 1 s4 2 stay
s3 0 s4 2 stay
s2 _ s4 _ stay
s2 2 s4 2 stay
s2 1 s4 1 stay
s2 0 s3 0 stay
```

Of course, it needs an input file. Given an input tape containing only the symbol 0, the output of the example will be 1. This example makes sure that the first if executes when the 0 is seen and that the second if does not execute now that the 0 has been overwritten by a 1.

Here's a more complicated test case (./test/if-if-else.tml) designed to make sure that programs with a dangling else are parsed the way they are supposed to be:

```
0,1,correct,incorrect

write correct
if (0)
   if (1)
      write incorrect
   else
      write incorrect

right
write 0
if (0)
   if (1)
      write incorrect
   else
      write correct
```

The correct output of executing this program is "correct correct". Here's the compiled machine:

```
s0 _ s1 correct stay
s0 incorrect s1 correct stay
s0 correct s1 correct stay
s0 1 s1 correct stay
s0 0 s1 correct stay
s3 _ s4 incorrect stay
s3 incorrect s4 incorrect stay
s3 correct s4 incorrect stay
s3 1 s4 incorrect stay
s3 0 s4 incorrect stay
s2 _ s5 _ stay
```

s2 incorrect s5 incorrect stay
s2 correct s5 correct stay
s2 1 s3 1 stay
s2 0 s5 0 stay
s5 _ s6 incorrect stay
s5 incorrect s6 incorrect stay
s5 correct s6 incorrect stay
s5 1 s6 incorrect stay
s5 0 s6 incorrect stay
s4 _ s6 _ stay
s4 incorrect s6 incorrect stay
s4 correct s6 correct stay
s4 1 s6 1 stay
s4 0 s6 0 stay
s1 _ s6 _ stay
s1 incorrect s6 incorrect stay
s1 correct s6 correct stay
s1 1 s6 1 stay
s1 0 s2 0 stay
s6 _ s7 _ right
s6 incorrect s7 incorrect right
s6 correct s7 correct right
s6 1 s7 1 right
s6 0 s7 0 right
s7 _ s8 0 stay
s7 incorrect s8 0 stay
s7 correct s8 0 stay
s7 1 s8 0 stay
s7 0 s8 0 stay
s10 _ s11 incorrect stay
s10 incorrect s11 incorrect stay
s10 correct s11 incorrect stay
s10 1 s11 incorrect stay
s10 0 s11 incorrect stay
s9 _ s12 _ stay
s9 incorrect s12 incorrect stay
s9 correct s12 correct stay
s9 1 s10 1 stay
s9 0 s12 0 stay
s12 _ s13 correct stay
s12 incorrect s13 correct stay
s12 correct s13 correct stay
s12 1 s13 correct stay
s12 0 s13 correct stay
s11 _ s13 _ stay
s11 incorrect s13 incorrect stay
s11 correct s13 correct stay

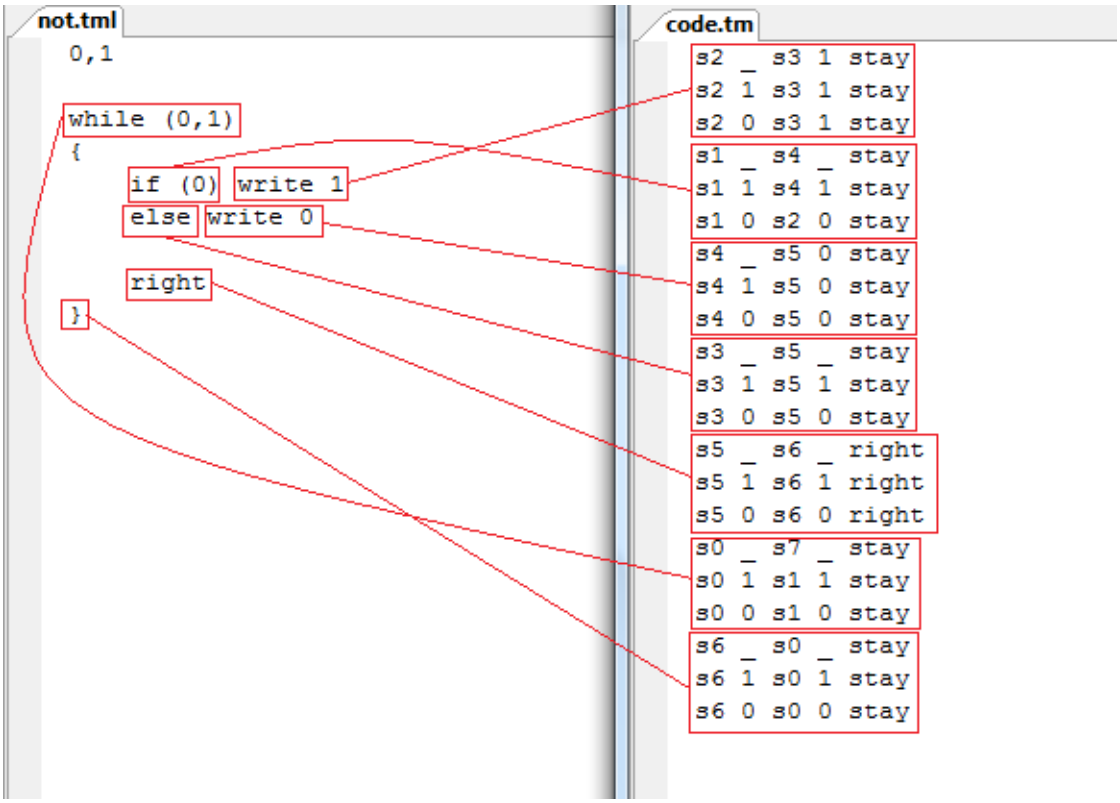
```
s11 1 s13 1 stay
s11 0 s13 0 stay
s8 _ s13 _ stay
s8 incorrect s13 incorrect stay
s8 correct s13 correct stay
s8 1 s13 1 stay
s8 0 s9 0 stay
```

The previous two test cases were designed to check the functionality of specific features. However, we also had several test cases that were designed to actually do something somewhat useful and use a variety of the language constructs. Here is one such example (./test/reverse.tml) that reverses reverses the binary string on its input tape:

```
0,1,x

while (0,1) right

write x

until (_)
{
   while (x) left

   if (0)
   {
      write x
      until(_) right
      write 0
   }
   else if (1)
   {
      write x
      until (_) right
      write 1
   }

   while (0,1) left
}

right

while (x)
{
   write _
   right
}
```

We'll spare you the compiled output for this example because it is rather long. In fact, we did not check the compiled output for the more complicated tests like this one manually, but rather used our test script to verify that it produced the correct output on the tape when executed.

Here's one more short example (which implements the bitwise NOT operator) that shows how the source language and compiled output correspond:



7. Lessons Learned

Isaac - I learned that thanks to O'Caml, writing a simple compiler from scratch doesn't need to be a painful exercise. In particular, generating the AST and doing the recursive walk of it is much nicer in O'Caml than in C. This was also the first non-trivial project I've ever written in a functional programming language, so I think the experience has broadened my programming skills somewhat.

Josh - O'Caml allows one to focus on the problem rather than the data structures necessary to its implementation. In particular, the elegance with which the parser and compiler are written is notable over comparable c-style code.

Keerti - Have a sound language design before starting coding. Learn the language(Ocaml) used for building the compiler well.
Ocaml is quite efficient and produces shorter and faster code compared when compared to languages like C.It was a good experience to understand all the nuances involved in language design with a flavor of functional programming.

Snehit - My main takeaway would be from the OCaml rather than TMSL. I think brevity at the expense of readability undoes a language's appeal. I still prefer LISP to OCaml : not just because it's closer in form and truer to lambda calculus, but also because it's more readable (in a really warped, shamelessly top-down sense).

Advice to future teams: begin early, focus on the language design prior to the implementation. Write test cases and regression tests in advance. Consider writing sample programs and discussing their results prior to writing a compiler to discover language related issues in advance. When designing your language, make sure you go through an example program and figure out how it would be compiled in detail by hand. If you can't figure out how to compile a specific example by hand, you will have a very hard time writing an algorithm to do the compilation in general for every possible program.

8. Appendix

Source Code

The code files are named using relative addressing to the project base directory.

---------- ./run.bat ----------

```
@echo off

if "%1" == "" goto error
if "%2" == "" goto error
if "%3" == "" goto error

.\compiler\compiler.exe %1 code.tm
.\machine\machine.exe code.tm %2 %3

echo Input tape:
type %2
echo.
echo Output tape:
type %3
echo.
echo Done - have a nice day. Thank you for using the run script(tm).
goto done

:error
echo "Usage: run.bat code.tmsl input.tape output.tape"
:done
```

---------- ./machine/tape_scanner.mll ----------

```
{ open Tape_parser }

rule token = parse
[' ' '\t' '\r' '\n'] { token lexbuf }
| ['a'-'z' 'A'-'Z' '0'-'9' '_']+ as symbol { SYMBOL(symbol) }
```

```
| eof { EOF }
```

---------- ./machine/scanner.mll ----------

```
{ open Parser }

rule token = parse
[' ' '\t' '\r' '\n'] { token lexbuf }
| "left" { LEFT }
| "right" { RIGHT }
| "stay" { STAY }
| ['a'-'z' 'A'-'Z' '0'-'9' '_']+ as id { ID(id) }
| eof { EOF }
```

---------- ./compiler/scanner.mll ----------

```
{ open Parser }

rule token = parse
| "if" { IF }
| "else" { ELSE }
| "unless" { UNLESS }
| "while" { WHILE }
| "until" { UNTIL }
| "left" { LEFT }
| "right" { RIGHT }
| "write" { WRITE }
| "exit" { EXIT }
| "else" { ELSE }
| '(' { LPAREN }
| ')' { RPAREN }
| '{' { LBRACE }
| '}' { RBRACE }
| (('_') | (['a'-'z''A'-'Z''0'-'9']+)) as lxm { SYMBOL(lxm) } (* blanks are special symbols which should never be concatenated *)
| ',' { COMMA }
| [' ' '\t' '\r' '\n'] { token lexbuf }
| "/*" { comment lexbuf }
| eof { EOF }
| _ as char { raise (Failure("illegal character " ^ Char.escaped char)) }
```

```
and comment = parse
"*/" { token lexbuf } (* we had issues ending comments with newline due to variet in operating systems so went with c-style comments insteads *)
| _ { comment lexbuf }
```

---------- ./machine/ast.mli ----------

```
type transition =
{
    current_state: string;
    input_symbol: string;
    next_state: string;
    output_symbol: string;
    move: int
}
```

---------- ./compiler/ast.mli ----------

```
type symbol = string

type stmt =                          (* Statements *)
  Block of stmt list                 (* { ... } *)
| If of symbol list * stmt           (* if (0,1) {} *)
| IfElse of symbol list * stmt * stmt
| Unless of symbol list * stmt
| While of symbol list * stmt        (* while (0,1) { left } *)
| Until of symbol list * stmt        (* until (0,1) { left } *)
| Left
| Right
| Write of symbol                    (* Write (0) *)
| Exit
| Program of symbol list * stmt list

type program = Program of symbol list * stmt list
```

---------- ./machine/machine.ml ----------

```
open Ast

exception Execution_halted

let rec follow_transition table state symbol = match table with
    [] -> raise Execution_halted
    | head::tail ->
        if (state, symbol) = (head.current_state, head.input_symbol) then
            (head.next_state, head.output_symbol, head.move)
        else
            follow_transition tail state symbol

let rec trim_blanks_front tape = match tape with
    [] -> []
    | head::tail -> if head = "_" then trim_blanks_front tail else head::tail

let trim_blanks tape =
    let tape = Array.to_list tape in
        Array.of_list (List.rev (trim_blanks_front (List.rev (trim_blanks_front tape))))

let write_output outfile tape = Array.iter (fun symbol -> output_string outfile symbol; output_char outfile ' ')
(trim_blanks tape)

let rec execute table tape state pos =
    (* print_string "execute state = "; print_string state; print_string " pos = "; print_int pos; print_newline ();
*)
    let symbol = try tape.(pos) with Invalid_argument err -> "_" in
    (* print_string "tape = "; write_output stdout tape; print_newline (); *)
    (* print_string "symbol = "; print_string symbol; print_newline (); print_newline (); *)
    try
        let (next_state, output_symbol, move) = follow_transition table state symbol in
        let next_pos = pos + move in
        try
            tape.(pos) <- output_symbol;
            execute table tape next_state next_pos
        with Invalid_argument err ->
            if pos < 0 then
            (
                assert (pos = -1);
                let next_pos = next_pos + 1 in      (* I'm appending to the front of the array, so all the
positions shift by 1 *)
```

```
                execute table (Array.append [| output_symbol |] tape) next_state next_pos
            )
            else
            (
                assert (pos = Array.length tape);
                execute table (Array.append tape [| output_symbol |]) next_state next_pos
            )
    with Execution_halted ->
        (* print_string "execution halted"; *)
        tape

let _ =
let lexbuf = Lexing.from_channel (open_in Sys.argv.(1)) in
let table = Parser.table Scanner.token lexbuf in
let lexbuf = Lexing.from_channel (open_in Sys.argv.(2)) in
let tape = Tape_parser.tape Tape_scanner.token lexbuf in
let tape = execute table (Array.of_list tape) "s0" 0 in
let outfile = open_out Sys.argv.(3) in
write_output outfile tape
```

---------- ./compiler/compiler.ml ----------

```
open Ast
open Printf

let outfile = open_out Sys.argv.(2)

let write_transition(current_state, input_symbol, next_state, output_symbol, move) =
    fprintf outfile "s%i %s s%i %s %s\n" current_state input_symbol next_state output_symbol move

exception Undefined_symbol of string

let check_symbol all_symbols symbol = if List.mem symbol all_symbols then () else raise
(Undefined_symbol symbol)

let check_symbol_list all_symbols symbol_list = List.iter (check_symbol all_symbols) symbol_list

let rec eval (symbols, state) = function
    Left -> List.iter ( fun sym -> write_transition(state, sym, state + 1, sym, "left") ) symbols; (symbols, state
+ 1)

    | Right -> List.iter ( fun sym -> write_transition(state, sym, state + 1, sym, "right") ) symbols; (symbols,
state + 1)
```

```
| Write(out_sym) -> check_symbol symbols out_sym;
                    List.iter (fun sym -> write_transition(state, sym, state + 1, out_sym, "stay") ) symbols;
                    (symbols, state + 1)

| Exit -> (symbols, state + 1)

| If(symbol_list, statements) ->
    check_symbol_list symbols symbol_list;
    let (symbols, next_state) = eval (symbols, state + 1) statements in
    List.iter (fun sym ->
            let next_state = if List.mem sym symbol_list then state + 1 else next_state in
            write_transition(state, sym, next_state, sym, "stay") )
        symbols;
    (symbols, next_state)

| IfElse(symbol_list, if_statements, else_statements) ->
    check_symbol_list symbols symbol_list;
    let (symbols, else_state) = eval (symbols, state + 1) if_statements in
    List.iter (fun sym ->
            let next_state = if List.mem sym symbol_list then state + 1 else else_state + 1 in
            write_transition(state, sym, next_state, sym, "stay") )
        symbols;
    let (symbols, next_state) = eval (symbols, else_state + 1) else_statements in
    List.iter (fun sym -> write_transition(else_state, sym, next_state, sym, "stay")) symbols;
    (symbols, next_state)

| Unless(symbol_list, statements) ->
    check_symbol_list symbols symbol_list;
    let (symbols, next_state) = eval (symbols, state + 1) statements in
    List.iter (fun sym ->
            let next_state = if List.mem sym symbol_list then next_state else state + 1 in
            write_transition(state, sym, next_state, sym, "stay") )
        symbols;
    (symbols, next_state)

| While(symbol_list, statements) ->
    check_symbol_list symbols symbol_list;
    let (symbols, jumpback_state) = eval (symbols, state + 1) statements in
    let next_state = jumpback_state + 1 in
    (* write transitions for condition test state *)
    List.iter (fun sym ->
            let next_state = if List.mem sym symbol_list then state + 1 else next_state in
            write_transition(state, sym, next_state, sym, "stay") )
        symbols;
    (* write transitions for jump-back state *)
    List.iter ( fun sym -> write_transition(jumpback_state, sym, state, sym, "stay") ) symbols;
```

```
        (symbols, next_state)


    | Until(symbol_list, statements) ->
        check_symbol_list symbols symbol_list;
        let (symbols, jumpback_state) = eval (symbols, state + 1) statements in
        let next_state = jumpback_state + 1 in
        (* write transitions for condition test state *)
        List.iter (fun sym ->
                let next_state = if List.mem sym symbol_list then next_state else state + 1 in
                write_transition(state, sym, next_state, sym, "stay") )
            symbols;
        (* write transitions for jump-back state *)
        List.iter ( fun sym -> write_transition(jumpback_state, sym, state, sym, "stay") ) symbols;
        (symbols, next_state)

    | Block(statements) -> List.fold_left eval (symbols, state) statements

    | _ -> (symbols, state) (* everything else is unimplemented. do nothing *)

let program = function
    Program(symbols, statements) -> let symbols = if List.mem "_" symbols then symbols else "_" :: symbols
in List.fold_left eval (symbols, 0) statements

let _ =
let lexbuf = Lexing.from_channel (open_in Sys.argv.(1)) in
program (Parser.program Scanner.token lexbuf)
```

```
----------  ./test/regression.ps1  -----------

"Running positive tests"

$sources = ls *.tml -name

$tests = 0
$successes = 0

foreach ($source in $sources)
{
   $tests++
   rm tmp.tm -ErrorAction SilentlyContinue
```

```
    rm tmp.output -ErrorAction SilentlyContinue
    ../compiler/compiler $source tmp.tm
    ../machine/machine tmp.tm ($source + ".input") tmp.output
    $diff = diff (get-content ($source + ".output")) (get-content tmp.output)
    if ($diff -ne $null)
    {
        write-error ("Test failed: " + $source)
        $diff
        echo ""
    }
    else
    {
        "Test succeeded: " + $source
        $successes++
    }
}

"Running negative tests"

$sources = ls *.neg -name

foreach ($source in $sources)
{
    $tests++
    rm error -ErrorAction SilentlyContinue
    ../compiler/compiler $source tmp.tm 2>error
    if ((get-content error) -match "error")
    {
        "Test succeeded: $source"
        $successes++
    }
    else
    {
        write-error "Test failed: $source"
    }

}



"$successes out of $tests succeeded"



----------  ./test/while.tml  ----------
```

```
0,1

while (0)
{
    write 1
    right
}
while (1)
{
    write 0
    right
}
```

---------- ./test/unless.tml ----------

```
0,1,2
write 1
unless (0){
    write 2
}
```

---------- ./test/multiply.tml ----------

```
/* multiplys two urnary numbers */
/* input form 0^m 1 0^n 1 */
/* output form 0^(m+n) */

0, 1, x

while (0)
{
    write _
    right
    until (1) right
    right

    /* begin copy */
    while (0)
    {
```

```
        write x
        until (_) right
        write 0
        until (x) left
        right
    }
    left
    while (x)
    {
        write 0
        left
    }
    /* end copy */

    left
    if (0)
    {
        until (_) left
        right
    }
    else
    {
        right
        write _
        until (1)
        {
            write _
            right
        }
        write _
    }
}
```

---------- ./test/if.tml ----------

```
0,1,2
if (0) write 1
if (0) write 2
```

---------- ./test/not.tml ----------

```
0,1

while (0,1)
{
    if (0) write 1
    else write 0

    right
}
```

---------- ./test/write010.tml ----------

```
0,1

write 0
right
write 1
right
left
right
write 0
```

---------- ./test/reverse.tml ----------

```
0,1,x

while (0,1) right

write x

until (_)
```

```
{
    while (x) left

    if (0)
    {
        write x
        until(_) right
        write 0
    }


else if (1)
    {
        write x
        until (_) right
        write 1
    }

    while (0,1) left
}

right

while (x)
{
    write _
    right
}
```

---------- ./test/unsignedsub.tml ----------

```
0,1
while (1) {
   if (1) {
        write _
        right
     while (1) {
          right
     }
        while (0) {
          right
        }
        if (1) {
```

```
            write 0
        }
        while (0,1) {
            left
        }
        right
    }
}
left
left
while (0,1) {
left
}
write 1
```

---------- ./test/until.tml ----------

```
0,1
until (1){
    write 1
    right
}
```

---------- ./test/unsignedadd.tml ----------

```
0,1
while (1){
    right
    if (0){
        right
        if (1){
            write 0
            left
            write 1
        }
    }
}
```

---------- ./test/if-if-else.tml ----------

```
0,1,correct,incorrect

write correct
if (0)
    if (1)
        write incorrect
    else
        write incorrect

right
write 0
if (0)
    if (1)
        write incorrect
    else
        write correct
```

---------- ./compiler/test/unsigned_subtraction.txt ----------

```
0,1
while (1) {
    if (1) {
        write _
        right
        while (1) {
            right
        }
        while (0) {
            right
        }
        if (1) {
            write 0
        }
        while (0,1) {
            left
        }
        right }
}
left
left
```

```
while (0,1) {
    left
}
write 1
```

---------- ./compiler/test/symbolsneg.txt ----------

```
_1,__,a_3_b

write _1_3
write
```

---------- ./compiler/test/symbol_list.txt ----------

```
1,2,3,4,5,6

if(1,2,3,4,5,6){}
```

---------- ./compiler/test/code_block.txt ----------

```
0, 1

if(0)
{
left
right
}
```

---------- ./compiler/test/symbols.txt ----------

```
1, 123, abc, a1
```

```
write 1
write _
write 123
write abc
write a1
```

---------- ./compiler/test/if.txt ----------

```
0,1,2
if (0) { write 1 }
if (0) { write 2 }
```

---------- ./compiler/test/while.txt ----------

```
0,1
write 0
while(0){
    write 1
}
```

---------- ./compiler/test/comments.txt ----------

```
symbol1

/* i'm a comment */
/* i'm a
multi
line comment
with keywords
WRITE
write l
left
*/
```

---------- ./compiler/test/example_from_lrm.txt ----------

```
while(1){
   if(1){
      write _
      while(1){
         right
      }
      while(0){
         right
      }
      if(1){
         write 0
      }
      while(0,1){
      left
      }
      right
   }
}
```

---------- ./compiler/test/stmt_list.txt ----------

```
1, 0

left
right
write 1
exit
if(0){}
while(0){}
exit
```

---------- ./compiler/test/reverse.txt ----------

```
0,1
```

```
while(0,1){
     if (0){
          write 1
          right
     }
     if (1){
          write 0
          right }
}
```

---------- ./compiler/test/whileneg.txt ----------

```
0,1
while()
{
write 1
}
```

---------- ./compiler/test/keywords.txt ----------

```
1, abc, 0

left
right
write 1
write _
write abc
exit
if(0){}
if(_){}
if(0,1,_){}
while(0){}
while(_){}
while(0,1,_){}
```

---------- ./compiler/test/exit.txt ----------

0,1
write 0
exit
write 1