All That Matrix

Final Project Report

8/22/2014
COMS W4115
Stefanie Zhou (sz2475)

Contents

1. Introduction

All That Matrix (ATM) is a programming language targeted at matrix manipulations with emphasize on clear syntax and lightweight compiler.

1.1 Motivation

Applications of matrix are very common across scientific fields. In statistics, matrices are used for probability calculations. In computer graphics, matrices are used to project and transform images. And you won't get through a lecture of linear algebra without encountering matrices.

ATM provides intuitive matrix related operators with the goal of avoiding as many built-in functions as possible and making it easy to write custom functions in the language itself. Thus, built-in types, operators, and keywords are kept to a minimal set.

1.2 Overview

The syntax of ATM is very similar to C and Java, so novice should have a minimal learning curve. ATM code is translated into a set of native bytecode, which then gets executed against a built-in stack to produce the output.

2. Language Tutorial

An ATM program is a single file consisting of functions, defined and written above the mandatory main function, which is where the program always kicks off.

2.1 First Example

This is greatest common divisor written in ATM. This example shows general purpose features in ATM including function declaration, while loop and conditionals.

```
gcd(a, b)
{
   while (a != b) {
     if (a > b) a = a - b;
     else b = b - a;
   }
   return a;
}

main()
{
  print(gcd(3,15));
}
```

2.2 Matrix Example
This example illustrates the declaration, initiation and accessing of matrix data types.

```
main()
{
  int i;
  int j;
  matrix[3][3] m;
  m = [1,2,3|4,5,6|7,8,9];
  for (i = 1 ; i < 4 ; i = i + 1) {
    for (j = 1 ; j < 4 ; j = j + 1) {
      print(m[i][j]);
    }
  }
}
```

2.3 Compile and Run Your Program
Write your code in a .atm file and compile it by running these two commands.

```
$ make
$ ./atm -c < [path to your .atm file]
```

This will compile and run your code. You can see the AST of your program by using the "-a" parameter and you can examine the list of bytecode generated by using the "-b" parameter.

3. Language Reference Manual

3.1 Introduction
All That Matrix is a programming language targeted at matrix manipulations with emphasize on the clear syntax and a lightweight compiler. All That Matrix provides intuitive matrix related operators with the goal of avoiding as many built-ins as possible and making it easy to write custom functions in the language itself. This language reference manual is inspired by the C reference manual [1].

3.2 Lexical Elements
3.2.1 Comments
Comments are delineated with an opening /* and closing */. The compiler will ignore comments. Nesting of comments is not supported.

```
/* This is a comment */
```

3.2.2 Identifiers

Identifiers are sequences of characters that must start with a lower case letter and can be followed by any number of upper-case letter, lower-case letters, digits, and underscores, used for naming variables and functions. Identifiers are case sensitive.

Identifier -> [a-z][a-zA-Z_0-9]*

3.2.3 Keywords

Keywords are reserved for use as part of the programming language and therefore, cannot be used for any other purposes.

int	matrix	main	return
if	else	for	while
export	print		

3.2.4 Punctuations

Parentheses are used to indicate function calls, signify conditionals, and group formal arguments to functions.

Curly Braces are used to indicate a block of statements.

Semicolons are used to signal the end of a statement and also to separate statements and expressions in for loops.

3.2.5 Constants

There are a total of four constants in ATM: integer literal, string literal, boolean, and matrix.

3.2.5.1 Integer Literals

An integer constant is a sequence of digits.

Integer Constant -> [0-9]+

3.2.5.2 String Literals

String literal constants are delineated by double quotation marks and can contain any character.

String Literal Constant -> “['a'-'z' 'A'-'Z' '0'-'9' '_' '.' ':' '-' ' ' '/']*”

3.2.5.4 Boolean

Boolean constants, used in conditional logic, are represented by integer literals: 1 for true and 0 for false.

Boolean -> 0 | 1

3.2.5.4 Matrix

Matrix constant are enclosed in square brackets with vertical bars separating the rows and commas separating the columns. Matrix constants are filled by integer literals.

[1, 2, 3 | 4, 5, 6] is a 2 by 3 matrix

3.3 Data Types

3.3.1 Int

Integers are used to represent boolean and to build compound type matrix. It must be declared before use.

```
int my_integer;
my_integer = 8;
```

3.3.2 Boolean

Booleans are represented by integers: 1 for true and 0 for false. Booleans are only intended to be used in conditionals, so they are not declared.

3.3.3 String

Strings are surrounded by double quotation marks and are only designed to be used in two places. The first one is in print statement such as

```
print(“test string”);
```

The second one is in specifying the export file name as in

```
export(out_val, “my_output.txt”);
```

3.3.4 Matrix

The one supported compound data types is matrix, which is declared with the keyword matrix and the number of rows and columns specified in brackets as in

```
matrix[3][4] my_matrix;
```

3.4 Expressions and Operators

3.4.1 Expressions

An expression consists of at least one operand and zero or more operators. Operands are one of the typed objects such as matrix and can be an identifier, a constant, or a function call that returns a value.

3.4.2 Binary Operators

Binary operators for int and matrix data types follow the standard arithmetic and matrix operation rules. These operators are valid between two objects of the same type for integers. However, for matrices, the types between the two expressions can differ for certain operators.

For example, multiplication between an integer and a matrix is equivalent to scaling the matrix by the integer, whereas multiplication between two matrices follows the standard matrix multiplication rules.

In other words, the behavior of the operators depends on the type of the operands provided. For example, when adding two integers: 5 +10, the result is 15. When adding two row matrices [a1, b1| c1, d1] + [a2, b2| c2, d2], the result is the matrix [a1+a2, b1+b2| c1+c2, d1+d2].

expression + expression
expression - expression
expression * expression

One additional operator for integers is division. Note that the result is rounded to integers according to the rules in OCaml.

expression / expression

3.4.3 Logical Operators

These logical operators between two integers or matrices evaluate to boolean and are to be used in control flow. The data type on the left and right sides of the operator must be the same. In the case of matrices, their dimension must be the same as well.

expression == expression
expression != expression

These are additional logical operators for expressions of integers only.

expression < expression
expression <= expression
expression > expression
expression >= expression

3.4.4 Built-in Features

All That Matrix also provides a limited set of built-in functions and features to retrieve and save information.

print() is a built-in function that print the item at the top of the stack. The output format for a matrix is spaces separating the columns and new lines separating the rows.

export(identifier, string filename) is a built-in function that writes the output to an external file specified.

col_count(matrix m) is a built-in function that returns the number of columns the input matrix has.

row_count(matrix m) is a built-in function that returns the number of rows the input matrix has.

3.5 Statements

All statements must end with a semi-colon. All statements either declare a variable, use, or modify an existing variable. If-then-else statements, for and while loops are supported. The syntax rules for them are the same as the C language. All of the following are examples of statements.

```
return 0;                          /* return statement */
If (a!=b)                          /* control flow */
foo(2);                            /* function call */
[2,4|2,4]*[1,0|8,2]                /* expression */
while (i > 0)                      /* while loop */
for (i = 1 ; i < n ; i = i + 1)    /* foo loop*/
```

3.6 Declarations

3.6.1 Program Definition

A program in All That Matrix consists of list of global variables and a list of functions. User-defined functions should be above the main function. The program always looks for the function main to start off.

3.6.2 Function Declarations

A function declaration must start with the name of the function, followed by a list of zero or more parameters separated by commas and enclosed in parenthesis. Functions in All That Matrix must be declared and implemented simultaneously. The result can be returned in a return statement. Nested functions are not supported.

```
function_name (type arg1, type arg2,…)
{
   function body
}
```

3.7 Scope

A declared object is only visible in the scope enclosed by the nearest curly bracket pair. Declarations made within functions are visible only within those functions. A declaration is not visible to declarations that came before it. An identifier declared outside of any curly bracket pairs is a global variable, and thus, is accessible from anywhere of the program.

3.8. References

[1] B. W. Kernighan and D. Ritchie. The C Programming Language, Second Edition. Precentice-Hall, 1988.

4. Project Plan

4.1 Process

Because this project was not a team project, I was responsible for all components. Hence, the approach I took may be somewhat different. I did not follow the approach where I did not move on to work on the parser until the scanner is completely done. I started with a very basic framework provided by Micro C and added new pieces to all components of the language iteratively.

There was no extensive and detailed period of project planning due to mostly timing constraint. I did not have all details of the language flushed out and I did not start coding until I've reviewed all lectures on Micro C and did some reading on O'Caml. By that time, it was already past mid of June, so I had only one month to turn over the project.

As I began writing the compiler, I realized that several rules I laid out initially was unclear and inconsistent and I had to go back and change it to make it work for the new specification.

However, development and Testing went well for me. I adopted the test-driven development approach where a new test was written before the code was in place to keep the development cycles short and focused. Core features are dealt with first before the built-in functions were included. Unit testing was the main focus until near the end of the development cycle, where integration testing kicked in.

4.2 Programming Style Guide

While this project did not run into the issue where different team members are vastly inconsistent in their coding style, I still try to adhere to the general style guide outlined in this section so that code across all components of the project are consistent and readable, which are the two main goals.

Spaces are used instead of tabs for indentation and grouping of blocks of similar structured code along with parenthesis. A single space should be placed on either side of assignment (=), operators (+, -, ...), and comparisons (>, < ...).

One blank line is used to separate different sections of the code, block comments and the code that follows it.

No line should be longer than 100 characters. It is recommended to put the condition and body of if statements on separate lines and use indentation and parenthesis. Exceptions can be made if the condition and the body are both really short. Compound statements (multiple statements on the same line separated by semicolons) are generally discouraged.

Comments are kept to a minimum. They should be descriptive, and not simply repeat what the code does.

4.3 Project Timeline

Start - End Date	Deliverables
- 06/11/2014	Proposal
- 07/02/2014	Language reference manual
- 07/31/2014	A very basic framework complete
08/01 – 08/20/2014	Short iterative development cycles for all components of the language
- 08/20/2014	Compile final report
- 08/22/2014	Project due

4.4 Project Log

Date	Focus
06/07/2014	Start thinking about the focus of the language
06/10/2014	Complete the project proposal
06/27/2014	Start defining the language rules
07/01/2014	Complete the language reference manual
07/19/2014	Complete all lectures on Micro C
07/23/2014	Read more about O'Camllex and O'Camlyacc
07/27/2014	Get a very basic framework ready for development
08/02/2014	Add core set of token rules to scanner and appropriate placeholders in parser, ast, bytecode, compiler, and executor
08/03/2014	Add customized matrix data type
08/09/2014	Work on local and global variables
08/10/2014	Work on function declaration and function calling
08/16/2014	Implement operators for matrix
08/17/2014	Add in built-in functions
08/18/2014	Start integration testing
08/20/2014	Start compiling the final report
08/22/2014	Complete the final report

* Note: each entry such as "work on …" and "implement" involve adding the appropriate pieces to the parser, ast, compiler, and executor to pass testing.

4.5 Development Environment

Programming language: O'Caml

Scanner: O'Camllex

Parser: O'Camlyacc

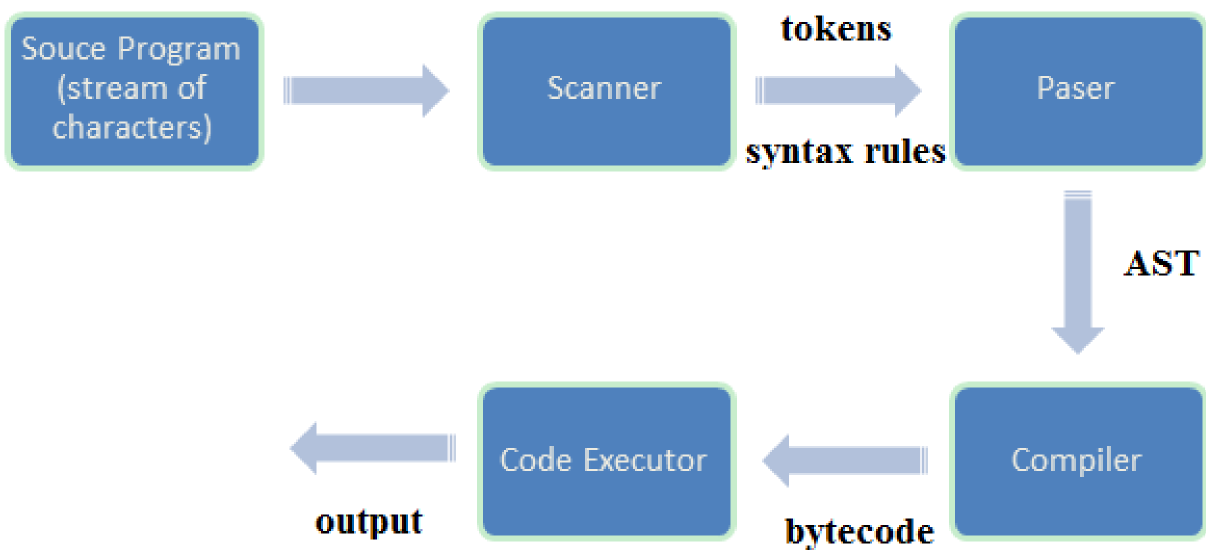
Test: automatic bash script for regression testing

Build: Makefile

5. Architectural Design

5.1 Overview

ATM is made up of a scanner, parser, AST, compiler and code executor.



5.2 Front-end

The scanner follows the basic convention of accepting the source file, converting it into a stream of tokens, eliminating useless tokens such as whitespace, comments, etc. The scanner raises an exception upon encountering of an illegal token.

The parser then accepts the token stream from the scanner and parses it based on the rules laid out in the language reference manual and constructs an abstract syntax tree. More useless tokens are eliminated in this process and an exception occurs when the input stream does not satisfy the predefined syntax.

5.3 Back-end

Instead of using translating the AST into Java code or some other language, I've decided to translate it into native bytecode which then gets executed off a stack I implemented to produce the final output. The stack is studied and taught in various computer science courses, but I've always only had its concept understood. Therefore, I decided to take this chance to implement it and get a first-hand experience on all the details behind it.

6. Testing Plan

6.1 Goal

There are two goals for testing. One is to decide what feature to implement next in the test-driven development. The other is to ensure that the new code does not introduce new bugs as in regression testing. These tests are not compressive, but they were created systematically, at least one test case for each portion of the language reference manual, in order to find any inconsistencies in the way data is treated.

6.2 Methods

A test suite was kept and maintained throughout the development phase of the project. Before a new feature was implemented, for example, global variables, at least one test case was written for it immediately. Hence, the development cycle was test-driven where the end of one development iteration is signaled by the passing of this new test case along with all other tests in the test suite for regression.

The test cases were kept small because they were designed to reduce debugging effort so that when a test fails, I will know which part, sometimes down to the exact byte code implementation, was causing the problem. Once added, no test case was ever deleted from the test suite.

Toward the end of the development cycle, longer and more compressive test cases were added to test the integration of the different components.

A test case consists of two files, (1) a .atm file which contains a program written in ATM, and (2) a .out file which contains the expected output. All test cases were contained within the /test directory. Automation in the form of a bash script was used for running all test cases in the test suite, comparing the actual output with the expected output, and logging the result in a text file.

6.3 Test Suite

file	focus
test-arith-int.atm	Add between integers
test-arith-int2.atm	Arithmetic between integers
test-arith-metric.atm	Add between matrices
test-arith-metric2.atm	Arithmetic between matrices
test-arith-metric3.atm	Arithmetic between integers and matrices
test-built-in-func.atm	built-in function col_count and row_count
test-built-in-func-export.atm	built-in function export with int
test-built-in-func-export2.atm	built-in function export with matrix
test-comment.atm	comment properly ignored
test-det.atm	built-in functions, variables, functions, if statement, operators
test-fib.atm	recursion, if statement, variables and function declarations
test-for1.atm	for loop with int
test-for2.atm	nested for loops with matrix
test-func1.atm	function declaration and call involving integers
test-func2.atm	function declaration and call involving matrices
test-func3.atm	function formal and actual arguments of type int
test-func4.atm	function formal and actual arguments of type matrix
test-gcd.atm	function declaration and call, while loop, if else statement
test-global1.atm	declaration and initialization of global variables of type int
test-global2.atm	declaration and initialization of global variables of type matrix
test-id.atm	identifiers

test-if1.atm	if statement – evaluates to true
test-if2.atm	if statement – evaluates to false
test-if3.atm	if else statement - evaluates to false
test-if4.atm	if else statement - evaluates to true
test-ops-int.atm	logical operators for type int
test-ops-metric.atm	logical operators for type matrix
test-print-string.atm	print of type string
test-var-int.atm	declaration and initialization of local variables of type int
test-var-metric.atm	declaration and initialization of local variables of type matrix
test-while.atm	while loop

6.4 Representative Test Case 1

This is one of the earliest test cases written for testing the initialization and declaration of local variable of local variables.

```
main()
{
   matrix[3][2] b;
   matrix[3][3] a;
   a = [1,2,3|4,5,6|7,8,9];
   b = [8,2,1|1,0,5];
   print(b);
   print(a);
}
```

This is the translated bytecode.

```
0 global variables
0 Jsr 2
1 Hlt
2 Ent 21
3 Lit 1
4 Lit 2
5 Lit 3
6 Lit 4
7 Lit 5
8 Lit 6
9 Lit 7
10 Lit 8
11 Lit 9
12 Lit 3
13 Lit 3
14 Max
15 Sfp 21
16 Drp
17 Lit 8
18 Lit 2
19 Lit 1
20 Lit 1
21 Lit 0
22 Lit 5
23 Lit 3
24 Lit 2
25 Max
26 Sfp 9
27 Drp
28 Lfp 9
29 Jsr -1
30 Drp
31 Lfp 21
32 Jsr -1
33 Drp
34 Lit 0
35 Rts 0
```

6.5 Representative Test Case 2

This is a test case targeted at testing the correct functionality of nested for loops.

```
main()
{
    int i;
    int j;
    matrix[3][3] m;
    m = [1,2,3|4,5,6|7,8,9];
    for (i = 1 ; i < 4 ; i = i + 1) {
      for (j = 1 ; j < 4 ; j = j + 1) {
        print(m[i][j]);
      }
    }
}
```

The bytecode produced for this is the following.

0 global variables 0 Jsr 2 1 Hlt 2 Ent 21 3 Lit 1 4 Lit 2 5 Lit 3 6 Lit 4 7 Lit 5 8 Lit 6 9 Lit 7 10 Lit 8 11 Lit 9 12 Lit 3 13 Lit 3 14 Max 15 Sfp 21 16 Drp 17 Lit 8	18 Lit 2 19 Lit 1 20 Lit 1 21 Lit 0 22 Lit 5 23 Lit 3 24 Lit 2 25 Max 26 Sfp 9 27 Drp 28 Lfp 9 29 Jsr -1 30 Drp 31 Lfp 21 32 Jsr -1 33 Drp 34 Lit 0 35 Rts 0

6.6 Representative Test Case 3

This is a more comprehensive test intended for integration testing.

```
is_square_matrix(input)
{
  return col_count(input) == row_count(input);
}
```

```
det2(input)
{
  return ((input[1][1])*(input[2][2])-(input[1][2])*(input[2][1]));
}

det3(input)
{
  int a;
  int b;
  int c;
  a = det2([input[2][2],input[2][3]|input[3][2],input[3][3]]);
  b = det2([input[2][1],input[2][3]|input[3][1],input[3][3]]);
  c = det2([input[2][1],input[2][2]|input[3][1],input[3][2]]);
  return (input[1][1])*a-(input[1][2])*b+(input[1][3])*c;
}

det(input)
{
  int ret_val;
  ret_val = 0;
  if (is_square_matrix(input)) {
    if (col_count(input)==2) {
      ret_val = det2(input);
    }
    if (col_count(input)==3) {
      ret_val = det3(input);
    }
  }
  return ret_val;
}

print_det(input)
{
  if (is_square_matrix(input)) {
    print("is a square matrix");
    print("determinant is");
    print(det(input));
  }
  else {
    print("is not a square matrix");
  }
}

main()
{
```

```
  matrix[2][2] test_matrix1;
  matrix[3][3] test_matrix2;
  matrix[2][3] test_matrix3;
  test_matrix1 = [2,8|1,7];
  test_matrix2 = [12,5,1|7,4,0|1,2,3];
  test_matrix3 = [12,5,1|7,4,0];
  print("test_matrix1 results:");
  print_det(test_matrix1);
  print("test_matrix2 results:");
  print_det(test_matrix2);
  print("test_matrix3 results:");
  print_det(test_matrix3);
}
```

The following is the bytecode produced for this.

0 global variables	116 Lit 2
0 Jsr 2	117 Acc
1 Hlt	118 Lfp -2
2 Ent 28	119 Lit 2
3 Lit 2	120 Lit 3
4 Lit 8	121 Acc
5 Lit 1	122 Lfp -2
6 Lit 7	123 Lit 3
7 Lit 2	124 Lit 3
8 Lit 2	125 Acc
9 Max	126 Lit 2
10 Sfp 7	127 Lit 2
11 Drp	128 Max
12 Lit 12	129 Jsr 199
13 Lit 5	130 Sfp 1
14 Lit 1	131 Drp
15 Lit 7	132 Lfp -2
16 Lit 4	133 Lit 1
17 Lit 0	134 Lit 2
18 Lit 1	135 Acc
19 Lit 2	136 Lfp -2
20 Lit 3	137 Lit 3
21 Lit 3	138 Lit 2
22 Lit 3	139 Acc
23 Max	140 Lfp -2
24 Sfp 19	141 Lit 1
25 Drp	142 Lit 3
26 Lit 12	143 Acc
27 Lit 5	144 Lfp -2
28 Lit 1	145 Lit 3

29 Lit 7 30 Lit 4 31 Lit 0 32 Lit 3 33 Lit 2 34 Max 35 Sfp 28 36 Drp 37 Stg test_matrix1 results: 38 Jsr -1 39 Drp 40 Lfp 7 41 Jsr 57 42 Drp 43 Stg test_matrix2 results: 44 Jsr -1 45 Drp 46 Lfp 19 47 Jsr 57 48 Drp 49 Stg test_matrix3 results: 50 Jsr -1 51 Drp 52 Lfp 28 53 Jsr 57 54 Drp 55 Lit 0 56 Rts 0 57 Ent 0 58 Lfp -2 59 Jsr 222 60 Beq 12 61 Stg is a square matrix 62 Jsr -1 63 Drp 64 Stg determinant is 65 Jsr -1 66 Drp 67 Lfp -2 68 Jsr 77 69 Jsr -1 70 Drp 71 Bra 4 72 Stg is not a square matrix 73 Jsr -1 74 Drp	146 Lit 3 147 Acc 148 Lit 2 149 Lit 2 150 Max 151 Jsr 199 152 Sfp 2 153 Drp 154 Lfp -2 155 Lit 1 156 Lit 2 157 Acc 158 Lfp -2 159 Lit 2 160 Lit 2 161 Acc 162 Lfp -2 163 Lit 1 164 Lit 3 165 Acc 166 Lfp -2 167 Lit 2 168 Lit 3 169 Acc 170 Lit 2 171 Lit 2 172 Max 173 Jsr 199 174 Sfp 3 175 Drp 176 Lfp -2 177 Lit 1 178 Lit 1 179 Acc 180 Lfp 1 181 Mul 182 Lfp -2 183 Lit 2 184 Lit 1 185 Acc 186 Lfp 2 187 Mul 188 Sub 189 Lfp -2 190 Lit 3 191 Lit 1

75 Lit 0 76 Rts 1 77 Ent 1 78 Lit 0 79 Sfp 1 80 Drp 81 Lfp -2 82 Jsr 222 83 Beq 22 84 Lfp -2 85 Jsr -3 86 Lit 2 87 Eql 88 Beq 6 89 Lfp -2 90 Jsr 199 91 Sfp 1 92 Drp 93 Bra 1 94 Lfp -2 95 Jsr -3 96 Lit 3 97 Eql 98 Beq 6 99 Lfp -2 100 Jsr 109 101 Sfp 1 102 Drp 103 Bra 1 104 Bra 1 105 Lfp 1 106 Rts 1 107 Lit 0 108 Rts 1 109 Ent 3 110 Lfp -2 111 Lit 2 112 Lit 2 113 Acc 114 Lfp -2 115 Lit 3	192 Acc 193 Lfp 3 194 Mul 195 Add 196 Rts 1 197 Lit 0 198 Rts 1 199 Ent 0 200 Lfp -2 201 Lit 1 202 Lit 1 203 Acc 204 Lfp -2 205 Lit 2 206 Lit 2 207 Acc 208 Mul 209 Lfp -2 210 Lit 2 211 Lit 1 212 Acc 213 Lfp -2 214 Lit 1 215 Lit 2 216 Acc 217 Mul 218 Sub 219 Rts 1 220 Lit 0 221 Rts 1 222 Ent 0 223 Lfp -2 224 Jsr -3 225 Lfp -2 226 Jsr -2 227 Eql 228 Rts 1 229 Lit 0 230 Rts 1

7. Lessons Learned

In this project, the one thing that I found most useful was the test suite. With many of the projects that I've done in the past, testing had never been woven into the development cycle as tightly as this project did. In the past, testing was done ad hoc with mostly random print statements and debugger at the most. However, the test-driven development I adopted this time put testing in the center of the project, and it helped discover all kinds of bugs.

Also, thinking in O'Caml was a very different experience for me, I spent a lot of time in the beginning to learn in detail how all the components work together from scanning of the input to translating it into AST and eventually, running the bytecode.

It was overwhelming initially, and I did not know how and where to start and the shortness of summer semester certainly did not help. I find myself procrastinating. I did not start the actual development until I've gone through all of the lectures on Micro C, which was past the middle of June. I was getting panicked as the deadline is fast approaching and had to cut certain features I had set out to do in the initial plan.

My advice is to start the project early, specifically, try to find ways to motivate your to work on the project. For people in a team, peer pressure can often work wonders. Set goals for each of the member the next time you meet. Allocate dedicated hours per week to work on the project as you would for lectures. I find that test-driven development worked very well for me because it made the development cycle shorter, made me focus on one thing at a time, and pushed me forward to the next iteration.

8. Appendix

8.1 scanner.mll

```
{ open Parser }

rule token = parse
   | [' ' '\t' '\r' '\n']  { token lexbuf }
   | "/*"     { comment lexbuf }
   | '|'     { BAR}
   | '('     { LPAREN }
   | ')'     { RPAREN }
   | '{'     { LBRACE }
   | '}'     { RBRACE }
   | ';'     { SEMI }
   | ','     { COMMA }
   | '+'     { PLUS }
   | '-'     { MINUS }
   | '*'     { TIMES }
   | '/'     { DIVIDE }
   | '='     { ASSIGN }
   | "=="    { EQ }
```

```
    | "!="     { NEQ }
    | '<'     { LT }
    | "<="    { LEQ }
    | ">"     { GT }
    | ">="    { GEQ }
    | "if"    { IF }
    | "else"  { ELSE }
    | "for"   { FOR }
    | "while" { WHILE }
    | "return" { RETURN }
    | "["     { LBRACKET }
    | "]"     { RBRACKET }
    | "int" as dt    { DATA_TYPE(dt) }
    | "matrix" as dt { DATA_TYPE(dt) }
    | ['0'-'9']+ as lxm { LITERAL(int_of_string lxm) }
    | ['a'-'z']['a'-'z' 'A'-'Z' '0'-'9' '_']* as lxm    { ID(lxm) }
    | ['\"']['a'-'z' 'A'-'Z' '0'-'9' '_' '.' ':' '-' ' ' '/']*['\"'] as str  { STRING(str) }
    | eof { EOF }
    | _ as char { raise (Failure("illegal character " ^ Char.escaped char)) }

and comment = parse
     "*/"  { token lexbuf }
    | _    { comment lexbuf }
```

8.2 parser.mly

```
%{ open Ast %}

%token SEMI LPAREN RPAREN LBRACE RBRACE LBRACKET RBRACKET COMMA
BAR
%token PLUS MINUS TIMES DIVIDE ASSIGN
%token EQ NEQ LT LEQ GT GEQ
%token RETURN IF ELSE FOR WHILE
%token <int> LITERAL
%token <string> ID
%token <string> STRING
%token <string> DATA_TYPE
%token EOF

%nonassoc NOELSE
%nonassoc ELSE
%right ASSIGN
%left EQ NEQ
%left LT GT LEQ GEQ
%left PLUS MINUS
%left TIMES DIVIDE
```

```
%start program
%type <Ast.program> program

%%

program:
   /* nothing */ { [], [] }
 | program vdecl { ($2 :: fst $1), snd $1 }
 | program fdecl { fst $1, ($2 :: snd $1) }

fdecl:
   ID LPAREN formals_opt RPAREN LBRACE vdecl_list stmt_list RBRACE
   { { fname = $1;
             formals = $3;
             locals = List.rev $6;
             body = List.rev $7 } }

formals_opt:
    /* nothing */ { [] }
  | formal_list   { List.rev $1 }

formal_list:
    ID                   { [$1] }
  | formal_list COMMA ID { $3 :: $1 }

vdecl_list:
    /* nothing */    { [] }
  | vdecl_list vdecl { $2 :: $1 }

vdecl:
    DATA_TYPE ID SEMI
    { {data_type = $1; id = $2; rows = 0; cols = 0} }
  | DATA_TYPE LBRACKET LITERAL RBRACKET LBRACKET LITERAL RBRACKET
ID SEMI
    { {data_type = $1; id = $8; rows = $3; cols = $6} }

stmt_list:
    /* nothing */  { [] }
  | stmt_list stmt { $2 :: $1 }

stmt:
    expr SEMI { Expr($1) }
  | RETURN expr SEMI { Return($2) }
  | LBRACE stmt_list RBRACE { Block(List.rev $2) }
  | IF LPAREN expr RPAREN stmt %prec NOELSE { If($3, $5, Block([])) }
```

```
  | IF LPAREN expr RPAREN stmt ELSE stmt    { If($3, $5, $7) }
  | FOR LPAREN expr_opt SEMI expr_opt SEMI expr_opt RPAREN stmt
      { For($3, $5, $7, $9) }
  | WHILE LPAREN expr RPAREN stmt { While($3, $5) }

expr_opt:
    /* nothing */ { Noexpr }
  | expr          { $1 }

expr:
    LITERAL          { Literal($1) }
  | ID               { Id($1) }
  | STRING           { String($1) }
  | expr LBRACKET expr RBRACKET LBRACKET expr RBRACKET { Access($1, $3, $6) }
  | LBRACKET matrix RBRACKET { Matrix($2) }
  | expr PLUS   expr { Binop($1, Add,   $3) }
  | expr MINUS  expr { Binop($1, Sub,   $3) }
  | expr TIMES  expr { Binop($1, Mult,  $3) }
  | expr DIVIDE expr { Binop($1, Div,   $3) }
  | expr EQ     expr { Binop($1, Equal, $3) }
  | expr NEQ    expr { Binop($1, Neq,   $3) }
  | expr LT     expr { Binop($1, Less,  $3) }
  | expr LEQ    expr { Binop($1, Leq,   $3) }
  | expr GT     expr { Binop($1, Greater,  $3) }
  | expr GEQ    expr { Binop($1, Geq,   $3) }
  | ID ASSIGN expr   { Assign($1, $3) }
  | ID LPAREN actuals_opt RPAREN { Call($1, $3) }
  | LPAREN expr RPAREN { $2 }

row:
    { [] }
  | expr { [$1] }
  | row COMMA expr { $3 :: $1 }

matrix:
    row { [List.rev $1] }
  | matrix BAR row { $1 @ [List.rev $3] }

actuals_opt:
    /* nothing */ { [] }
  | actuals_list  { List.rev $1 }

actuals_list:
    expr                    { [$1] }
  | actuals_list COMMA expr { $3 :: $1 }
```

8.3 ast.ml

```
type op = Add | Sub | Mult | Div | Equal | Neq | Less | Leq | Greater | Geq

type expr =
    Literal of int
  | String of string
  | Id of string
  | Access of expr * expr * expr
  | Matrix of expr list list
  | Binop of expr * op * expr
  | Assign of string * expr
  | Call of string * expr list
  | Noexpr

type stmt =
    Block of stmt list
  | Expr of expr
  | Return of expr
  | If of expr * stmt * stmt
  | For of expr * expr * expr * stmt
  | While of expr * stmt

type var_decl = {
    data_type : string;
    id : string;
    rows: int;
    cols: int
}

type func_decl = {
    fname : string;
    formals : string list;
    locals : var_decl list;
    body : stmt list;
}

type program = var_decl list * func_decl list

let rec string_of_expr = function
    Literal(l) -> string_of_int l
  | String(s) -> s
  | Id(s) -> s
  | Access(m, r, c) -> string_of_expr m ^ " " ^ string_of_expr r ^ " " ^ string_of_expr c
  | Matrix(m) -> String.concat ", " (List.map string_of_expr (List.concat m))
  | Binop(e1, o, e2) -> string_of_expr e1 ^ " " ^ (
      match o with
```

```
          Add -> "+" | Sub -> "-" | Mult -> "*" | Div -> "/"
     | Equal -> "==" | Neq -> "!="
     | Less -> "<" | Leq -> "<=" | Greater -> ">" | Geq -> ">=") ^ " " ^ string_of_expr e2
  | Assign(v, e) -> v ^ " = " ^ string_of_expr e
  | Call(f, el) -> f ^ "(" ^ String.concat ", " (List.map string_of_expr el) ^
     String.concat ", " (List.map (fun e -> "1") el) ^ ")"
  | Noexpr -> ""

let rec string_of_stmt = function
    Block(stmts) -> "{\n" ^ String.concat "" (List.map string_of_stmt stmts) ^ "}\n"
  | Expr(expr) -> string_of_expr expr ^ ";\n";
  | Return(expr) -> "return " ^ string_of_expr expr ^ ";\n";
  | If(e, s, Block([])) -> "if (" ^ string_of_expr e ^ ")\n" ^ string_of_stmt s
  | If(e, s1, s2) ->  "if (" ^ string_of_expr e ^ ")\n" ^
     string_of_stmt s1 ^ "else\n" ^ string_of_stmt s2
  | For(e1, e2, e3, s) -> "for (" ^ string_of_expr e1  ^ " ; " ^ string_of_expr e2 ^ " ; " ^
     string_of_expr e3  ^ ") " ^ string_of_stmt s
  | While(e, s) -> "while (" ^ string_of_expr e ^ ") " ^ string_of_stmt s

let string_of_vdecl var =
  var.data_type ^ " " ^ var.id ^ ";\n"

let string_of_fdecl fdecl =
  fdecl.fname ^ "(" ^ String.concat ", " fdecl.formals ^ ")\n{\n" ^
  String.concat "" (List.map string_of_vdecl fdecl.locals) ^
  String.concat "" (List.map string_of_stmt fdecl.body) ^
  "}\n"

let string_of_program (vars, funcs) =
  String.concat "" (List.map string_of_vdecl vars) ^ "\n" ^
  String.concat "\n" (List.map string_of_fdecl funcs)
```

8.4 bytecode.ml

```
type bstmt =
    Lit of int         (* Push a literal *)
  | Stg of string      (* Push a string *)
  | Max                (* Indicate matrix data type *)
  | Acc                (* Access matrix*)
  | Drp                (* Discard a value *)
  | Bin of Ast.op      (* Perform arithmetic on top of stack *)
  | Lod of int         (* Fetch global variable *)
  | Str of int         (* Store global variable *)
  | Lfp of int         (* Load frame pointer relative *)
  | Sfp of int         (* Store frame pointer relative *)
  | Jsr of int         (* Call function by absolute address *)
```

```
  | Ent of int          (* Push FP, FP -> SP, SP += i *)
  | Rts of int          (* Restore FP, SP, consume formals, push result *)
  | Beq of int          (* Branch relative if top-of-stack is zero *)
  | Bne of int          (* Branch relative if top-of-stack is non-zero *)
  | Bra of int          (* Branch relative *)
  | Hlt                 (* Terminate *)

type prog = {
    num_globals : int;   (* Number of global variables *)
    size_globals : int;  (* Size of global variables *)
    text : bstmt array;  (* Code for all the functions *)
  }

let string_of_stmt = function
    Lit(i) -> "Lit " ^ string_of_int i
  | Stg(s) -> "Stg " ^ s
  | Max -> "Max"
  | Acc -> "Acc"
  | Drp -> "Drp"
  | Bin(Ast.Add) -> "Add"
  | Bin(Ast.Sub) -> "Sub"
  | Bin(Ast.Mult) -> "Mul"
  | Bin(Ast.Div) -> "Div"
  | Bin(Ast.Equal) -> "Eql"
  | Bin(Ast.Neq) -> "Neq"
  | Bin(Ast.Less) -> "Lt"
  | Bin(Ast.Leq) -> "Leq"
  | Bin(Ast.Greater) -> "Gt"
  | Bin(Ast.Geq) -> "Geq"
  | Lod(i) -> "Lod " ^ string_of_int i
  | Str(i) -> "Str " ^ string_of_int i
  | Lfp(i) -> "Lfp " ^ string_of_int i
  | Sfp(i) -> "Sfp " ^ string_of_int i
  | Jsr(i) -> "Jsr " ^ string_of_int i
  | Ent(i) -> "Ent " ^ string_of_int i
  | Rts(i) -> "Rts " ^ string_of_int i
  | Bne(i) -> "Bne " ^ string_of_int i
  | Beq(i) -> "Beq " ^ string_of_int i
  | Bra(i) -> "Bra " ^ string_of_int i
  | Hlt    -> "Hlt"

let string_of_prog p =
  string_of_int p.num_globals ^ " global variables\n" ^
  let funca = Array.mapi
      (fun i s -> string_of_int i ^ " " ^ string_of_stmt s) p.text
  in String.concat "\n" (Array.to_list funca)
```

8.5 compile.ml

```
open Ast
open Bytecode

module StringMap = Map.Make(String)

(* Symbol table: Information about all the names in scope *)

type env = {
    function_index : int StringMap.t; (* Index for each function *)
    global_index   : int StringMap.t; (* Address for global variables *)
    local_index    : int StringMap.t; (* FP offset for locals *)
    arg_index      : int StringMap.t; (* FP offset for args *)
}

(* val enum : int -> 'a list -> (int * 'a) list *)

let rec enum stride n = function
    [] -> []
  | hd::tl -> (n, hd) :: enum stride (n+stride) tl

let rec enum_vars n = function
    [] -> []
  | hd::tl -> (
      if hd.data_type="matrix" then
        ((n+3+(hd.rows*hd.cols)), hd.id)
      else
        (n+1, hd.id)) :: enum_vars
        (if hd.data_type="matrix" then (n+3+(hd.rows*hd.cols)) else (n+1)) tl

(*helper function for calculating the size of allocated variables*)

let size_vars =
  function vars  -> List.fold_left (
    fun s l ->
      s + (if l.data_type = "matrix" then
      (3+(if l.rows > 0 then l.rows else 1)*(if l.cols > 0 then l.cols else 1))
    else 1)) 0 vars

(* val string_map_pairs StringMap 'a -> (int * 'a) list -> StringMap 'a *)

let string_map_pairs map pairs =
  List.fold_left (fun m (i, n) -> StringMap.add n i m) map pairs

(* Translate a program in AST form into a bytecode program.  Throw an
   exception if something is wrong, e.g., a reference to an unknown
```

```
  variable or function *)

let translate (globals, functions) =

  (* Allocate "addresses" for each global variable *)

  let global_indexes = string_map_pairs StringMap.empty (enum_vars 0 globals) in

  (* Assign indexes to function names; built-in "print" is special *)
  let built_in_functions =
    StringMap.add "export" (-4) (
    StringMap.add "col_count" (-3) (
    StringMap.add "row_count" (-2) (
    StringMap.add "print" (-1) StringMap.empty))) in

  let function_indexes = string_map_pairs built_in_functions
      (enum 1 1 (List.map (fun f -> f.fname) functions)) in

  (* Translate a function in AST form into a list of bytecode statements *)

  let translate env fdecl =

    (* Bookkeeping: FP offsets for locals and arguments *)

    let num_formals = List.length fdecl.formals
    and size_locals = size_vars fdecl.locals
    and local_offsets = enum_vars 0 fdecl.locals
    and formal_offsets = enum (-1) (-2) fdecl.formals in

    let env = { env with
      local_index = string_map_pairs StringMap.empty local_offsets;
      arg_index = string_map_pairs StringMap.empty formal_offsets} in

    let rec expr = function
            Literal i -> [Lit i]
      | String s -> [Stg (String.sub s 1 (String.length s - 2))]
      | Id s -> (
          try [Lfp (StringMap.find s env.local_index)]
          with Not_found -> try [Lfp (StringMap.find s env.arg_index)]
          with Not_found -> try [Lod (StringMap.find s env.global_index)]
          with Not_found -> raise (Failure ("undeclared variable " ^ s)))
      | Access(m, r, c) -> expr m @ expr c @ expr r @ [Acc]
      | Matrix(m) -> List.concat (List.map expr (List.concat m)) @
                [Lit (List.length (List.nth m 0))] @ [Lit (List.length m)] @ [Max]
      | Binop (e1, op, e2) -> expr e1 @ expr e2 @ [Bin op]
      | Assign (s, e) -> expr e @ (
```

```
      try [Sfp (StringMap.find s env.local_index)]
      with Not_found -> try [Sfp (StringMap.find s env.arg_index)]
           with Not_found -> try [Str (StringMap.find s env.global_index)]
           with Not_found -> raise (Failure ("undeclared variable " ^ s)))
    | Call (fname, actuals) -> (
      try (List.concat (List.map expr (List.rev actuals))) @
           [Jsr (StringMap.find fname env.function_index) ]
      with Not_found -> raise (Failure ("undefined function " ^ fname)))
    | Noexpr -> []

  in let rec stmt = function
          Block sl     -> List.concat (List.map stmt sl)
    | Expr e       -> expr e @ [Drp]
    | Return e     -> expr e @ [Rts num_formals]
    | If (p, t, f) -> let t' = stmt t and f' = stmt f in
                  expr p @ [Beq(2 + List.length t')] @
                  t' @ [Bra(1 + List.length f')] @ f'
    | For (e1, e2, e3, b) -> stmt (Block([Expr(e1);
                         While(e2, Block([b; Expr(e3)]))]))
    | While (e, b) -> let b' = stmt b and e' = expr e in
                  [Bra (1+ List.length b')] @ b' @ e' @
                  [Bne (-(List.length b' + List.length e'))]

  in [Ent size_locals] @      (* Entry: allocate space for locals *)
  stmt (Block fdecl.body) @  (* Body *)
  [Lit 0; Rts num_formals]   (* Default = return 0 *)

in let env = { function_index = function_indexes;
            global_index = global_indexes;
            local_index = StringMap.empty;
            arg_index = StringMap.empty } in

(* Code executed to start the program: Jsr main; halt *)

let entry_function = try
  [Jsr (StringMap.find "main" function_indexes); Hlt]
with Not_found -> raise (Failure ("no \"main\" function")) in

(* Compile the functions *)

let func_bodies = entry_function :: List.map (translate env) functions in

(* Calculate function entry points by adding their lengths *)

let (fun_offset_list, _) = List.fold_left
    (fun (l,i) f -> (i :: l, (i + List.length f))) ([],0) func_bodies in
```

```
  let func_offset = Array.of_list (List.rev fun_offset_list) in

  { num_globals = List.length globals;
    size_globals = size_vars globals;
    (* Concatenate the compiled functions and replace the function
       indexes in Jsr statements with PC values *)
    text = Array.of_list (List.map (function
           Jsr i when i > 0 -> Jsr func_offset.(i)
      | _ as s -> s) (List.concat func_bodies))
  }
```

8.6 execute.ml

```
open Ast
open Bytecode

(* Stack layout just after "Ent":

              <-- SP
   Local n
   ...
   Local 0
   Saved FP   <-- FP
   Saved PC
   Arg 0
   ...
   Arg n *)

let execute_prog prog =
  let stack = Array.make 1024 "0"
  and globals = Array.make prog.size_globals "0" in

  let rec exec fp sp pc = match prog.text.(pc) with
    Lit i  -> stack.(sp) <- string_of_int i ; exec fp (sp+1) (pc+1)
  | Stg s  -> stack.(sp) <- s ; exec fp (sp+1) (pc+1)
  | Max    -> stack.(sp) <- "Max"; exec fp (sp+1) (pc+1)
  | Acc    -> if stack.(sp-3) = "Max" then
               let rows = (int_of_string stack.(sp-4)) and
                   cols = (int_of_string stack.(sp-5)) and
                   r = (int_of_string stack.(sp-1)) and
                   c = (int_of_string stack.(sp-2)) in (
               stack.(sp-5-(rows*cols)) <- stack.(sp-6-(rows*cols)+((r-1)*cols)+c);
               exec fp (sp-4-(rows*cols)) (pc+1))
  | Drp    -> exec fp (sp-1) (pc+1)
  | Bin op -> if stack.(sp-1) = "Max" || stack.(sp-2) = "Max" then ((
              match op with
```

```
        Add | Sub | Equal | Neq -> (
          let rows = (int_of_string stack.(sp-2)) and
              cols = (int_of_string stack.(sp-3)) in (
          for i = 4 to (3+rows*cols) do
            let op1 = (int_of_string stack.(sp-3-i-rows*cols)) and
                op2 = (int_of_string stack.(sp-i)) in
            stack.(sp-i) <- (let boolean i = if i then "1" else "0" in
            match op with
              Add   -> string_of_int (op1 + op2)
            | Sub   -> string_of_int (op1 - op2)
            | Equal -> boolean (op1 = op2)
            | Neq   -> boolean (op1 != op2))
          done))
      | Mult  -> (
          if stack.(sp-1) <> "Max" then (
            let rows = (int_of_string stack.(sp-3)) and
                cols = (int_of_string stack.(sp-4)) and
                const = (int_of_string stack.(sp-1)) in (
            stack.(sp-1) <- "Max";
            stack.(sp-2) <- string_of_int rows;
            stack.(sp-3) <- string_of_int cols;
            for i = 5 to (4+rows*cols) do
              stack.(sp-i+1) <- string_of_int (const*(int_of_string stack.(sp-i)))
            done))
          else (
            let rows = (int_of_string stack.(sp-2)) and
                cols = (int_of_string stack.(sp-3)) in (
            if stack.(sp-rows*cols-4) <> "Max" then (
              for i = 4 to (3+rows*cols) do
                stack.(sp-i) <- string_of_int (int_of_string
                stack.(sp-rows*cols-4)*(int_of_string stack.(sp-i)))
              done)
            else (
              let rows2 = (int_of_string stack.(sp-5-rows*cols)) and
                  cols2 = (int_of_string stack.(sp-6-rows*cols)) in (
              if cols2 != rows then (raise (Failure(
                "Operators for * do not satisfy matrix multiplication criteria")))
              else (
                let ops1 = ref [] and ops2 = ref [] and
                    res = ref [] and sum = ref 0 in (
                let count = ref 0 and
                    m = (Array.sub stack (sp-rows*cols-6-rows2*cols2) (rows2*cols2)) in
                for i = 0 to (Array.length m - 1) do
                  count := (!count + 1);
                  ops1 := (!ops1 @ [Array.get m i]);
                  if !count = cols2 then (
```

```
          count := 0;
          let len = List.length !ops1 in
          for j = cols2 downto 1 do
            ops1 := (!ops1 @ [List.nth !ops1 (len - j)])
        done)
      done;
      let count = ref 0 and
          m = (Array.sub stack (sp-rows*cols-3) (rows*cols)) in (
      for r = 1 to rows2 do
        for c = 1 to cols do
          for i = 0 to (Array.length m - 1) do
            count := (!count + 1);
            if !count = c then (ops2 := (!ops2 @ [Array.get m i]));
            if !count = cols then (count := 0)
          done
        done
      done;
      count := 0;
      stack.(sp-2) <- string_of_int rows2;
      for i = 0 to (List.length !ops1 - 1) do
        count := (!count + 1);
        sum := (!sum + ((int_of_string (List.nth !ops1 i))*(
                int_of_string (List.nth !ops2 i))));
        if !count = cols2 then (
          res := (!res @ [!sum]); count := 0; sum := 0;)
        done;
        for i = 0 to (List.length !res - 1) do
          stack.(sp-3-(rows2*cols)+i) <- (string_of_int (List.nth !res i))
        done)
    )))))
  )));
  (match op with
  | Add | Sub | Mult -> ()
  | Equal -> let rows = (int_of_string stack.(sp-2)) and
               cols = (int_of_string stack.(sp-3)) in
           stack.(sp-1) <- (let cmp = (
             Array.fold_left (fun s e -> s + (int_of_string e)) 0 (
               Array.sub stack (sp-3-rows*cols) (rows*cols))) in
             if cmp = (rows*cols) then "1" else "0")
  | Neq -> let rows = (int_of_string stack.(sp-2)) and
             cols = (int_of_string stack.(sp-3)) in
         stack.(sp-1) <- (let cmp = (
           Array.fold_left (fun s e -> s + (int_of_string e)) 0 (
             Array.sub stack (sp-3-rows*cols) (rows*cols))) in
           if cmp > 0 then "1" else "0"));
  exec fp sp (pc+1))
```

```
            else (
              let op1 = (int_of_string stack.(sp-2)) and
                  op2 = (int_of_string stack.(sp-1)) in (
              stack.(sp-2) <- (let boolean i = if i then "1" else "0" in
              match op with
                Add     -> string_of_int (op1 + op2)
              | Sub     -> string_of_int (op1 - op2)
              | Mult    -> string_of_int (op1 * op2)
              | Div     -> string_of_int (op1 / op2)
              | Equal   -> boolean (op1 =  op2)
              | Neq     -> boolean (op1 != op2)
              | Less    -> boolean (op1 <  op2)
              | Leq     -> boolean (op1 <= op2)
              | Greater -> boolean (op1 >  op2)
              | Geq     -> boolean (op1 >= op2));
              exec fp (sp-1) (pc+1)))
| Lod i   -> if globals.(i-1) = "Max" then
               let rows = (int_of_string globals.(i-2)) and
                   cols = (int_of_string globals.(i-3)) in (
               for j = 1 to (3+rows*cols) do
                  stack.(sp+(3+rows*cols)-j) <- globals.(i-j)
               done;
               exec fp (sp+(rows*cols+3)) (pc+1))
             else (
               stack.(sp) <- globals.(i-1);
               exec fp (sp+1) (pc+1))
| Str i   -> if stack.(sp-1) = "Max" then
               let rows = (int_of_string stack.(sp-2)) and
                   cols = (int_of_string stack.(sp-3)) in (
               for j = 1 to (3+rows*cols) do
                  globals.(i-j) <- stack.(sp-j)
               done)
             else (globals.(i-1) <- stack.(sp-1));
             exec fp sp (pc+1)
| Lfp i   -> if i < 0 then let rec f1 = (fun x offset ->
               if offset = (-2) then x
               else if stack.(fp+x) = "Max" then
                 f1 (x-3-(int_of_string stack.(fp+x-1))*(
                   int_of_string stack.(fp+x-2))) (offset+1)
                 else f1 (x-1) (offset+1)) in (
               if stack.(fp+(f1 (-2) i)) = "Max" then
                 let rows = (int_of_string stack.(fp+(f1 (-2) i)-1)) and
                     cols = (int_of_string stack.(fp+(f1 (-2) i)-2)) in (
                 for j = 1 to (rows*cols+3) do
                   stack.(sp+(rows*cols+3)-j) <- stack.(fp+(f1 (-2) i)-j+1)
                 done;
```

```
          exec fp (sp+rows*cols+3) (pc+1))
        else (
          stack.(sp) <- stack.(fp+i);
          exec fp (sp+1) (pc+1)))
      else if stack.(fp+i-1) = "Max" then (
        let rows = (int_of_string stack.(fp+i-2)) and
            cols = (int_of_string stack.(fp+i-3)) in (
        for j = 1 to (rows*cols+3) do
          stack.(sp+(rows*cols+3)-j) <- stack.(fp+i-j)
        done;
        exec fp (sp+(if stack.(fp+i-1) = "Max" then (rows*cols+3) else 1)) (pc+1)))
      else (
        stack.(sp) <- stack.(fp+i);
        exec fp (sp+1) (pc+1))
| Sfp i   -> if stack.(sp-1) = "Max" then (
        let rows = (int_of_string stack.(sp-2)) and
            cols = (int_of_string stack.(sp-3)) in (
        for j = 1 to (rows*cols+3) do
          stack.(fp+i-j) <- stack.(sp-j)
        done))
      else (stack.(fp+i) <- stack.(sp-1));
      exec fp sp (pc+1)
| Jsr(-1) -> if stack.(sp-1) = "Max" then (
        let rows = (int_of_string stack.(sp-2)) and
            cols = (int_of_string stack.(sp-3)) in (
        for i = rows downto 1 do
          Array.iter (fun e -> Printf.printf "%s " e)
            (Array.sub stack (sp-3-i*cols) cols);
          Printf.printf "\n"
        done))
      else (print_endline stack.(sp-1));
      exec fp sp (pc+1)
| Jsr(-2) -> if stack.(sp-1) = "Max" then
        let rows = (int_of_string stack.(sp-2)) and
            cols = (int_of_string stack.(sp-3)) in (
        stack.(sp-3-(cols*rows)) <- string_of_int rows;
      exec fp (sp-2-(cols*rows)) (pc+1))
| Jsr(-3) -> if stack.(sp-1) = "Max" then
        let rows = (int_of_string stack.(sp-2)) and
            cols = (int_of_string stack.(sp-3)) in (
        stack.(sp-3-(cols*rows)) <- string_of_int cols;
      exec fp (sp-2-(cols*rows)) (pc+1))
| Jsr(-4) -> let oc = open_out stack.(sp-1) in (
        if stack.(sp-2) = "Max" then
          let rows = (int_of_string stack.(sp-3)) and
              cols = (int_of_string stack.(sp-4)) in (
```

```
          for i = rows downto 1 do
            Array.iter (fun e -> Printf.fprintf oc "%s " e) (
            Array.sub stack (sp-4-i*cols) cols);
            Printf.fprintf oc "\n"
        done)
        else Printf.fprintf oc "%s" stack.(sp-2));
        exec fp sp (pc+1)
| Jsr i   -> stack.(sp) <- string_of_int (pc + 1);
        exec fp (sp+1) i
| Ent i   -> stack.(sp) <- string_of_int fp;
        exec sp (sp+i+1) (pc+1)
| Rts i   -> let j = (if i > 0 then (let rec f1 = (
          fun x offset ->
            if offset = 0 then x
            else if stack.(fp+x) = "Max" then
              f1 (x-3-(int_of_string stack.(fp+x-1))*(
                int_of_string stack.(fp+x-2))) (offset+1)
             else f1 (x-1) (offset+1)) in (f1 (-2) (-i)))
          else i) in (
        let new_fp = int_of_string stack.(fp) and
            new_pc = int_of_string stack.(fp-1) in
        if stack.(sp-1) = "Max" then (
          let rows = (int_of_string stack.(sp-2)) and
              cols = (int_of_string stack.(sp-3)) in
          for x = 1 to (rows*cols+3) do
            stack.(fp-j-x) <- stack.(sp-x)
          done)
        else stack.(fp-j-1) <- stack.(sp-1);
        exec new_fp (fp-j) new_pc)
| Beq i   -> exec fp (sp-1) (pc + if (int_of_string stack.(sp-1)) =  0 then i else 1)
| Bne i   -> exec fp (sp-1) (pc + if (int_of_string stack.(sp-1)) != 0 then i else 1)
| Bra i   -> exec fp sp (pc+i)
| Hlt     -> ()

in exec 0 0 0
```

8.7 atm.ml

```
type action = Ast | Bytecode | Compile

let _ =
let action =
  if Array.length Sys.argv > 1 then
    List.assoc Sys.argv.(1) [("-a", Ast);
                             ("-b", Bytecode);
                             ("-c", Compile)]
```

```
  else Compile in
  let lexbuf = Lexing.from_channel stdin in
  let program = Parser.program Scanner.token lexbuf in match action with
    Ast -> let listing = Ast.string_of_program program in print_string listing
  | Bytecode -> let listing = Bytecode.string_of_prog (Compile.translate program) in
print_endline listing
  | Compile -> Execute.execute_prog (Compile.translate program)
```