String **C**omputation Progr**am**

Final Report

Scott Pender
scp2135@columbia.edu

COMS4115 Fall 2012

12/20/2012

Table of Contents

1 Introduction

1.1 Mission

Design and implement a custom language whose compiler is built using O'Caml. The language should be able to solve a problem by accepting a combination of commands, algorithms, and data, perform appropriate computation, and then return meaningful output. Additionally, this goal should be achieved using a minimalistic base language.

1.2 Language Description

String **C**omputation Progr**am** (**scam**) is a small, yet powerful compiled language created for the sole purpose of text processing. Using a minimal character set, scam allows a program to interpret and process string tokens. Adapting a hand-chosen blend of features from PERL, Python, and awk scripting languages, scam uses shorthand notation to denote relationships between strings. Key components of the scam language include variables, arrays, functions, control statements, strings, and integers.

Simply put, scam may be used to compute algorithms on string data. The language itself does not include any algorithms to process this data beyond Boolean comparison operators. Rather than include processing rules in the language itself, scam provides the ability to access included data structures and their respective attributes so that you as the developer can create your own rules.

1.3 Syntax

The syntax of scam makes an easy task of developing a program to perform string manipulations. The tab-delimited functions and control statements allow for programs that are both quick to develop an easy to read. Many of the operators used in the language will be familiar to developers including the '=' assignment operator and common control statements including if and which. Overall, scam is a comfortable and intuitive language to develop in that produces efficient and useful code.

2 Tutorial

The String Computation Language is, as its name indicates, a string computation language that allows users to develop quick code that can efficiently translate and manipulate input text.

2.1 Start with MAIN

Every program that is executed begins execution with the MAIN function, a common paradigm among today's programming languages. MAIN does not accept arguments from the command line and must be explicitly declared as 'MAIN()'. A sample hello world program is listed below.

```
MAIN()
        a = "Hello"
        b = "world"
        @ = a + " " + b
```

2.2 Functions

Many programs will require more than just a single function call. Using the recursive processing capabilities of scam, functions may call themselves or any other defined function (independent of declaration order). Function names may include any combination of letters and integers and may intermix capitalization at will (except for MAIN).

```
MAIN()
        a = "A"
        PrintA(A)
```

```
Print(a)
        @ = a
```

2.3 Control Statements and Booleans

Scam provides the IF and WHILE control statements for handling conditional operations. This may be useful for array iteration or condition output. In both cases, the block of statements is executed after a Boolean condition is met. In scam 0 is false and every other number (positive or negative) is evaluated as true. There is no explicit "true" or "false" keyword in scam. It is important to note that a Boolean evaluation must result in an integer, so the sum of the parts of any complex expression used must evaluate as an integer value. A sample usage of the WHILE conditional statement is included below.

```
MAIN()
        x = 2
        y = " test "
        WHILE(x>0)
                @=y
                x = x - 1
```

2.4 Arrays, Strings, and Integers

Scam is not intended to be used for any real mathematical computation, and intentionally only includes the '+' and '-' operators for arithmetic on integers. The purpose of including them at all is for Boolean evaluation (as mentioned earlier) and for referencing an index of the subset of a string or array. Types may be explicitly set using the type indicators int, str, and str[] but do not have to be if you wish to keep a variable's type mutable.

2.4.1 A Note on Printing

Scam allows the printing of string values only. Integers may never be output and there is no conversion operation for an integer. If there is a need to output an integer value it must be defined as a string from the beginning. Similarly, arrays may not be printed directly. If a part or whole of the array must be printed, it must be done through array reference. A sample use of integers, arrays, and strings in a printing capacity is shown below that also makes use of the length function to get the length of an array.

```
MAIN()
        y = "4"
        str[] x = ["1","2","3",y]
        int len = |x|
        WHILE(len)
                @ = x[len]
                len = len -1
```

2.5 Running a Program

In order to execute a scam program a developer must use the following syntax in a linux/unix-based operating system:

```
./scam [-a|-b|-c][-d] < program_name.scam
```

The following options are available (mostly for debugging purposes):

-a (ast)	Print the abstract syntax tree generated during parsing
-b (bytecode)	Print the bytecode representation generated during compilation

-c (compile) Compile and execute the program (default mode if no flag is set)

-d (debug) Compile and execute the program with a printout of the machine state as each bytecode operation is executed.

3 Reference Manual

3.1 Lexical Conventions

There are four kinds of tokens: identifiers, keywords, expression operators, and other separators.

3.1.1 Identifiers (Names)

An identifier may consist of a sequence of letters and digits. Special characters are not allowed. Identifiers are used to specify function and variable names. Reserved keywords may not be used as identifiers.

3.1.2 Keywords

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

FOREACH	int	MAIN
WHILE	str	
IF	str[]	

3.1.3 Comments

The ? character denotes a single-line comment. The ? must be used independently, outside of quotation marks. Multi-line comments are not available.

3.1.4 Line Terminators

The line feed character is used to terminate a line. More simply, there is no formal line termination character; lines are terminated automatically when a new line is fed to the parser.

3.1.5 Whitespace Characters

As noted previously the line feed character (\n) is used to denote line termination. The tab character (\t) denotes ownership of statements to the last statement that used one less tab character. This is used in cases of control statements and functions where the statements to be executed are grouped with by indentation. Control statements may be nested within functions, but functions may never be nested (only function calls). The space character ‘ ‘ is generally ignored, though it may be required to separate otherwise adjacent identifiers.

```
EX:     myFunction(X,Y)
                IF(1<2)
                        @=”Hello world”
                @=”If they are strings, the concatenation of X & Y is ” + X + Y
```

3.2 Scope

3.2.1 Programs

A program must be entirely self-contained and does not allow interaction with other programs directly.

3.2.2 Functions

Functions may be called by any other function independent of function definition order. There are no access modifiers.

3.2.3 Variables
Variables are not inherited and there are no global variables. A variable must be defined explicitly within each function as either a parameter or local variable.

3.3 Programs
Each program in scam consists of a document containing one MAIN function and zero to many helper functions.

3.4 Functions
A function is the definition of a sequence of statements to be executed in a specific order. Functions may accept input variables and return a value. Functions may also be called recursively.

3.4.1 Function definition
The following example demonstrates a function definition. The method signature is for a function 'myLabel' that accepts parameters X,Y, and Z. Note that Z must be an integer in this definition. Tab is used as the delimiter to denote the definition of a function. Within the function definition, the function label may be set like a variable to indicate a return value. The default return value is "". Since this is set like a variable, it can be used to make recursive calls to itself.

```
Ex:     myLabel(X, Y, int Z)
                ? Impl of myLabel
                myLabel = "optional returnValue"
```

3.4.2 Function call
Functions may be called from within any other function in a program. The example below demonstrates a function call.

```
Ex:     myLabel (X, Y, Z)
```

Implicit return from function (can return string or integer) can be set as value of another variable or sent to output (console, file

```
Ex:     @ = myLabel (X,Y,Z)
```

3.5 Variables
Variables are dynamically typed. A variable can be set to point to an integer, array, or string. A variable can later be re-defined to be of a different type. However, variables must still be referenced and defined properly according to the rules for each type. Type reference issues may be solved using static typing through explicit type enforcement. Variables are set using the '=' character with the variable identifier on the left-hand-side and the intended value on the right-hand-side. The following scenarios demonstrate variable definitions and their resulting values.

- X = 2
  - Define variable X as an integer value of 2
- X = "2"
  - Define variable X as string value of "2"
- XY="strinng"
  - Define variable XY as string "strinng"
- XY = []
  - Define XX as an empty array
- |X|
  - Length of value in X.
    - If string, return string length

        - If array, return array length
        - If integer, return integer value
        - Default return value for undefined variables is 0

3.6 Types

3.6.1 Strings

Strings, the primary focus of the scam language, may be compiled into an array of strings and may themselves be referenced as an array of (individual character) strings. Strings may be composed of any characters except for quotes themselves (or string formatting characters).

Stings are surrounded by quotes (“ ”) at all times. Everything between an opening and closing quote is part of that string. There is no need to escape characters within a string as everything within quotes is valid except for string formatting characters. String formatting characters, ~N for new line and ~T for tab character, may not be escaped.

String values may be concatenated using the string concatenation operator.

Whenever a string value is evaluated as null or is yet undefined, the strings value will be “” (empty string).

Since a string is itself an array, a string’s length, value at an index, and substring may be referenced in the same manner as an array.

3.6.2 Arrays of Strings

An array of strings is effectively a two-dimensional array since a string is an array itself. However, an array may not be defined as having more than one dimension. Arrays are only defined for string values, numbers are not permitted. Index order must be sequential and starts at 0.

Since array variables may continuously be re-assigned it is possible (though computationally costly) to construct an empty array, create a new array that contains one more element, and assign the original variable containing the array, the value of the new array. This could then be repeated iteratively using the array concatenation operator.

3.6.2.1 Array Definition

- X = []
    - Empty array of length 0.
- M = [X,Y,Z]
    - Create an array where M[0]=X, M[1]=Y, & M[2]=Z. If more than one array value is specified in this manner, all values MUST be string values.
- R = [x]
    - If x is an integer this will yield an array of size x; if x is a string this will yield a single-celled array containing the string value of x.
- M[0] = “mystring”
    - Explicitly define the value at an index.
- M[0] = X[1]
    - Set the value at an index equal to the value of another array’s cell. The referenced cell appears as a string value.
- M=X
    - Set one array to the value of another array. Overwrite existing array.

3.6.2.2 Array Reference

- X[indexVal]

        - Returns cell of index value, previously defined as a variable. If referenced cell has no value, return "".
  - X[0:2]
    - Represents the sub-array containing the first 3 cells.
  - X
    - The name of the array returns entire array.
  - |X|
    - Return the array length.
  - Reference to non-existent cell (outside array bounds) also yields "".

3.6.3 Integer

Integers [0-9] are used in the context of array manipulation. Mathematical operations are very simplistic, limited to addition and subtraction operations. Boolean true and false are represented by the integer values 1 and 0 respectively.

To obtain the string equivalent of an integer the integer must be surrounded by quotes ("9").

Since a negative array index or length may not be derived or used, a negative integer value serves no true purpose that coincides with the purpose of scam. As a result, there is no negative (-) unary operator defined for integers.

3.6.4 Explicit types

Any variable set to the value of a type listed previously may be re-assigned to another type without conflict as long as references to that variable are appropriate to the new type. In some circumstances we would like to avoid the potential conflict while referencing a variable by explicitly setting a variable's type. Note that this qualifier may also be used in function parameters to enforce types for input. The following qualifiers are provided immediately prior to a variable's identifier.

3.6.4.1 int <var_name>

Integer type.

3.6.4.2 str <var_name>

String type.

3.6.4.3 str[]<var_name>

String array type.

3.7 Expressions

3.7.1 Generic expression conventions

Notation described here may be used on all types.

3.7.1.1 (expression)

Parentheses are used to group expressions. Nested parentheses are allowed. Strings and arrays may only use parenthesis with equality operators.

3.7.1.2 Equality operators

Equality operators group left to right. Equality operators are valid for any type (string, array, or integer) and are used to compare value. Arrays must have the same size and all values in corresponding indices must be equal to be considered equal. Equality operators yield 0 if the specified relation is false and 1 if it is true.

3.7.1.2.1 expression == expression

3.7.1.2.2 expression != expression

3.7.2 Concatenation

The ‘+’ symbol has an entirely different meaning in the context of strings and arrays than it does in the context of integer algebraic operations.

3.7.2.1 string + string (String Concatenation)

Two or more strings or variables with string values may be appended together. When the resulting string is formed, the input strings are joined from left to right.

```
Ex:     myStr = “Hello”+ “ ” + “World!”
        @=myStr
        → # Hello World!
```

3.7.2.2 array + array (Array Concatenation)

Two or more arrays or variables with array values may be appended together. When the resulting array is formed, arrays are added from left to right with the leftmost array’s contents starting at the 0 index, effectively following the same pattern as string concatenation.

3.7.3 Integer Operators

The following operators may only be used when all operands are integers. Note that many of these operators consume or return Boolean values and that in scam Boolean values are represented by the integers 1 and 0.

3.7.3.1 Unary Operators

Expressions with unary operators group right to left. Unary operators yield 0 if the specified relation is false and 1 if it is true.

3.7.3.1.1 !expression

Exclamation point denotes the NOT operator. True if the expression is false (0).

3.7.3.2 Relational operators

The relational operators group left to right. Relational operators yield 0 if the specified relation is false and 1 if it is true.

3.7.3.2.1 expression < expression

3.7.3.2.2 expression > expression

3.7.3.2.3 expression <= expression

3.7.3.2.4 expression >= expression

3.7.3.3 expression || expression

A double pipe symbol denotes the OR operator. True if either expression is true (1). If an integer value is anything other than 0 it will evaluate to true.

3.7.3.4 expression && expression

A double ampersand denotes the AND operator. True if both expressions are true (1). If an integer value is anything other than 0 it will evaluate to true.

3.7.3.5 Integer Algebra
The following functions have specific behavior defined for integers that differs from the behavior defined for strings and arrays. Scam follows standard conventions for integer algebraic evaluation with operators grouped left to right.

3.7.3.5.1 expression + expression

3.7.3.5.2 expression – expression

3.8 Input/Output

3.8.1 Output

3.8.1.1 Output X to console (concatenates all string values, available for strings only)
- @= X

3.8.1.2 Output to file (available in scam 2.0!)
- @filepath = "text to print to file"
- @"C:/my folder/textfile" = "text to print to file"

3.8.2 Input (available in scam 2.0!)

3.8.2.1 From console
- X = @

3.8.2.2 From file
- X = @filepath
- X = @"./my folder/filename.txt"

3.8.2.3 From argument in MAIN
- X = @1

3.9 Control Statements
Pre-defined labels using the same notation as a standard function. The evaluated expression will accept 0 as false (unless ! is used). All other values are true.

3.9.1 IF(X)
```
IF (X)
    Do a bunch of stuff
```

3.9.2 WHILE(X)
```
WHILE(X)
    Do a bunch of stuff
```

4 Project Plan

4.1 Process

The project plan used was to apply a very simple waterfall approach and get the front-end working befor the backend. While an iterative approach would have been much more efficient to weed out bugs and prevent the inevitable mass of errors (once I discovered what would not I work in the program execution I had to re-work earlier stages), the waterfall approach was the only option due to limitations on time and understanding of the implementation details.

4.2 Style Guide

The following basic principles of coding style were used by all members of my team:

- Avoid external libraries where possible – it makes the code less portable.
- Uses comments everywhere possible to explain what you are doing. It can help when others look at your code and it can help when you take a large break between development of components.
- Unless it becomes impossible, develop to the spec, not your own tendencies
- Use a functional style wherever possible
- Indent and space your code properly so it is readable and easy to find for debugging/review purposes.
- K.I.S.S. – Keep the code as simple and brief as possible
- Use functions everywhere. If you are writing code more than once, you are writing it too many times.
- Wrap everything in exception handlers during testing/development. Isolate the error handling to a single module before deployment.
- If all else fails, just try to make it work.

4.3 Timeline

	September				October				November				December		
Task / Week	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Proposal	🟦	🟦	🟦												
Reference Manual				🟦	🟦	🟦	🟦	🟦							
Ast							🟦	🟦	🟦	🟦	🟦	🟦			
Compiler														🟦	🟦
Bytecode														🟦	🟦
Bytecode Interpreter														🟦	🟦
Testing															🟦

4.4 Roles / Responsibilities

Developer/Manager/Tester(limited) : Scott Pender

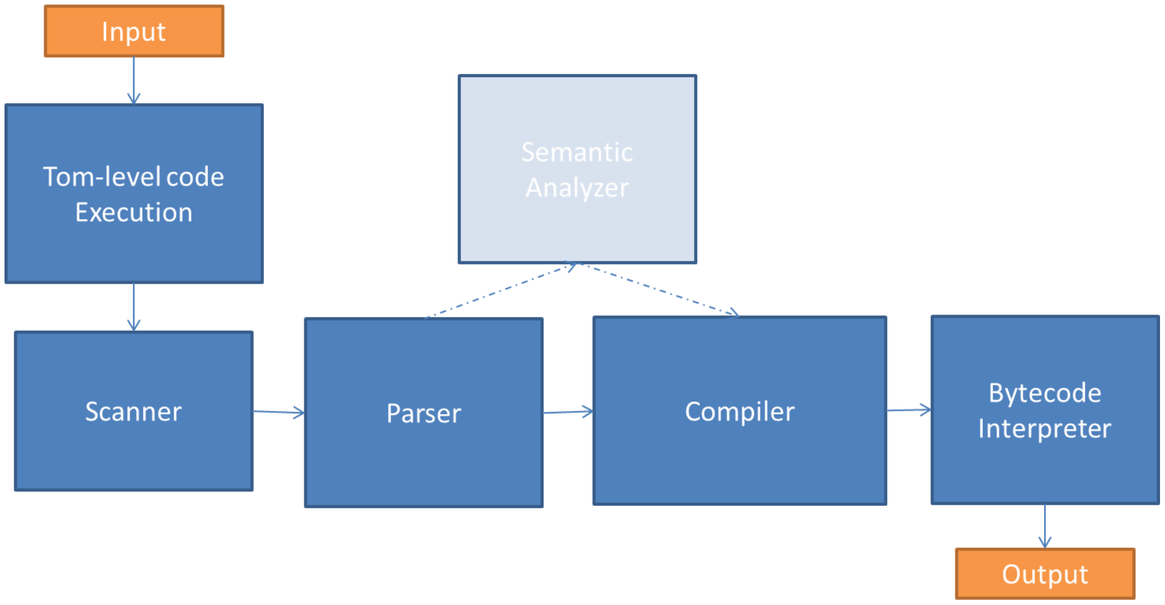
4.5 Software Development Environment

I mostly use an Ubuntu VM with a combination of the base terminal and gedit. After some recent internet issues I have switched to a Mac using the base terminal and TextWrangler for code writing.

Good code was used for code hosting and versioning (http://code.google.com/p/string-computation-program/) – while it is not needed for one person, I always trust a cloud before a VM. Also, I really wanted to try out using GIT as I had not used it before.

5 Architectural Design

An input program is fed directly into the top-level program. With the help of the Bytecode and Ast definition files code is handed off to the Scanner, then the Parser, then the Compiler, then the Bytecode Interpreter. A Semantic Analyzer was prepared to handle type resolution and enforcement, but there was not enough time to deploy a final version in the distributed code. I implemented ALL of the controls.



6 Test Plan

There are a number of errors in the programs execution. Much of this is a result of little testing being completed. A lack of time, a lack of team resources, and poor project management limited the test window to about 2 days. Currently much of it stands untested. The test code is not significant enough to include since I had very little time for testing. A bash script was prepared to test a large number of cases simultaneously, though it was rarely used.

6.1 Sample Program

A sample of a small program in scam is provided below.

```
? This is a program to compare files called “diff”
? The program is called using the following command:
? “scam diff filename1 filename2”  (without quotes)

MAIN()
        data1 = @1
        data2 = @2
        DIFF(data1, data2)
        @ = “Diff test complete”

DIFF(file1, file2)
        lineNum = 0
        file1Unfinished = 1
        file2Unfinished = 1
        WHILE(file1Unfinished OR file2Unfinished)
                IF(lineNum > file1)
                        file1Unfinished = 0
                        @ = “DIFF = Line ” + lineNum  + “of ” + @2 + “ does not exist in ” + @1
                IF(lineNum > file2)
                        file2Unfinished = (0)
                        @ = "DIFF = Line " + lineNum + " of " + @1 + " does not exist in " + @2
                IF(file1Unfinished AND file2Unfinished)
                        IF(file1[lineNum] != file2[lineNum])
                                COMPARE_LINES(file1(lineNum), file2(lineNum)), lineNum)
                lineNum = lineNum + 1

COMPARE_LINES(line1, line2, lineNum)
        charNum = 0
        testFinished = 0
        WHILE(!testFinished)
                IF(line1[charNum]== line2[charNum])
                        @ = "DIFF = Character " + charNum + " of line " + lineNum + " does not match"
                IF((charNum >= line1) OR (charNum >= line2))
                        testFinished = 1
                charNum = charNum + 1
```

7 Lessons Learned

The biggest lesson learned here is to constantly re-evaluate your timelines, requirements, and realistic goals to achieve in a final product. In this project I did not re-evaluate my options until the very last minute at which point I has already spent entirely too much time on components that did not function properly. I did not clearly establish the requirements – surely a one person team will produce a smaller product, but to what extent and where should my focus have been? Advice for future students: work with a team, collaborate oftern and early, and PLAN PLAN PLAN. Also, get a watch – time is key.

8 Appendix

8.1 Scanner.ml

```
{ open Parser }

rule token = parse
[' ' '\r' '\n'] {token lexbuf} (* Whitespace *)
| '?'           {comment lexbuf} (* Comments *)
| '\"' ([^'\"']+ as str) '\"' {STRING(str)} (* strings *)
| '\t'          {TAB}
(**| '\t'+'\n' {TABENDL} (* allow tabs at the end of a line *)*)
| '('           {LPAREN}
| ')'           {RPAREN}
| '['           {LBRACK}
| ']'           {RBRACK}
| '+'           {PLUS}
| '-'           {MINUS}
| "=="          {EQ}
| "!="          {NEQ}
| '<'           {LT}
| '>'           {GT}
| "<="          {LEQ}
| ">="          {GEQ}
| "&&"          {AND}
| "||"          {OR}
| '!'           {NOT}
| ','           {COMMA}
| ':'           {COLON}
| '|'           {BAR}
| '='           {ASSIGN}
| "FOREACH"     {FOREACH}
| "foreach"     {FOREACH}
| "WHILE"       {WHILE}
| "while"       {WHILE}
| "IF"          {IF}
| "if"          {IF}
| "str"         {STR}
| "str[]"       {ARR}
| "int"         {INT}
| '@'           {IO}
| ['0'-'9']+ as lxm { INTEGER(int_of_string lxm) }
| ['a'-'z' 'A'-'Z' '0'-'9']+ as lxm { ID(lxm) }
| eof { EOF }
| _ as char { raise (Failure("illegal character " ^ Char.escaped char)) }

and comment = parse
'\n' {token lexbuf}
| _    { comment lexbuf }
```

8.2 Parser.mly

```
%{open Ast

let parse_error s = (* Called by the parser function on error *)
       print_endline s;
       flush stdout

let stab = ref []

let print_stab  s = print_endline "stab = ";
       List.iter (fun v -> print_string (Ast.string_of_var v)) s

%}

%token TAB TABENDL LPAREN RPAREN LBRACK RBRACK COMMA COLON BAR IO
%token PLUS MINUS ASSIGN
%token AND OR LT GT LEQ GEQ EQ NEQ NOT
%token RETURN FOREACH WHILE IF
%token INT STR ARR
%token <int> INTEGER
%token <string> ID
%token <string> STRING
%token EOF

%nonassoc END
%nonassoc TAB TABENDL
%right INT STR ARR
%right ASSIGN
%right NOT IO
%left EQ NEQ
%left LT GT LEQ GEQ
%left PLUS MINUS
%left AND OR

%start program
%type <Ast.program> program

%%

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

fdecl: /* function declaration */
       ID LPAREN params_opt RPAREN stmt_list
       {{fname = $1;
        params = $3;
        body = List.rev $5;
        locals = List.rev !stab}}
```

```
params_opt: /* function parameters */
       /* nothing */ {stab:=[]; [] } /* arbitrary place to reset table for ea function */
       | params_list   { stab:=[]; List.rev $1 }

params_list: /* build function parameters as list */
       vdecl                   { [$1] }
       | params_list COMMA vdecl { $3 :: $1 }

vdecl:
       INT ID {{varType = Ast.Int; varName = $2;  dType = Ast.Int}}
       | STR ID {{varType = Ast.Str; varName = $2; dType = Ast.Str}}
       | ARR ID {{varType = Ast.Arr; varName = $2; dType = Ast.Arr}}
       | ID {{varType = Ast.Any; varName = $1; dType = Ast.Any}}

stmt_list:
       /* nothing */  { [] }
       | stmt_list TAB stmt { $3 :: $1 }
/*
stmt_tabopt:
       TABENDL {}    _tabopt
       |  {}
*/
stmt:
       RETURN expr { Return($2) }
       | IF LPAREN expr RPAREN stmt_list %prec END  { If($3, Block(List.rev $5)) }
       | WHILE LPAREN expr RPAREN stmt_list %prec END  { While($3, Block(List.rev $5)) }
       /*| IO term ASSIGN expr {Output($2, $4)}      id, string;  any */
       | IO ASSIGN expr {Output(Null, $3)}   /* any */
       | vdecl ASSIGN expr    {stab := $1::!stab; Assign($1, $3)}
       | vdecl                {stab := $1::!stab; Assign($1, Term(Null))}
       | ID LBRACK INTEGER RBRACK ASSIGN expr {SetAIndex(Id($1), Integer($3), $6)}

expr:
       term    {Term($1)}
       | LPAREN expr RPAREN{ $2 }
       | ID LPAREN args_opt RPAREN { Call($1, $3) }
       | ID LBRACK term COLON term RBRACK {Range(Id($1),$3,$5)} /* int,id; int,id */
       /* Binops */
       | expr PLUS   term { Binop($1, Add,   $3) } /* int, string, id, array */
       | expr MINUS  term { Binop($1, Sub,   $3) }  /* int, id */
       | expr EQ     term { Binop($1, Eq, $3) }  /* string, array, id, int */
       | expr NEQ    term { Binop($1, Neq,   $3) }  /* string, array, id, int */
       | expr LT     term { Binop($1, Lt,  $3) }  /* int, id */
       | expr GT     term { Binop($1, Gt,  $3) }  /* int, id */
       | expr LEQ    term { Binop($1, Leq,   $3) }  /* int, id */
       | expr GEQ    term { Binop($1, Geq,   $3) }  /* int, id */
       | expr AND    term { Binop($1, And,   $3) }  /* int, id */
       | expr OR     term { Binop($1, Or,   $3) }  /* int, id */
       | ID LBRACK term RBRACK {Binop(Term(Id($1)), Vat, $3)}  /* int, id */
       /* Unops */
       | NOT expr             {Unop($2,Not)} /* resolves to int */
       /*| IO term                    {Unop(Term($2),In)}  /* string, id, int */
       | BAR term BAR         {Unop(Term($2),Len)}

term:
       ID                     { Id($1) }
       | INTEGER              { Integer($1) }
       | STRING               { String($1) }
       | LBRACK term_opt RBRACK{ Array($2) } /* int, vars, strings */

args_opt:
       /* nothing */ { [] }
       | args_list  { List.rev $1 }

args_list:
       expr                    { [$1] }
```

```
        | args_list COMMA expr { $3 :: $1 }

term_opt:
        /* nothing */ { [] }
        | term_list  { List.rev $1 }

term_list:
        term                    { [$1] }
        | term_list COMMA term { $3 :: $1 }
```

8.3 Ast.ml

```
type op = Add | Sub | Eq | Neq | Lt | Gt | Leq | Geq | And | Or | Vat | Len | In | Out | Not | Rng | Sai
type typ = Int | Arr | Str | Any

type term =
        Integer of int
        | Id of string
        | String of string
        | Array of term list
        | Null

type var = {
        varType : typ;
        varName : string;
        mutable dType  : typ;
}

type expr =
        Term of term
        | Call of string * expr list
        | Range of term * term * term
        | Binop of expr * op * term
        | Unop of expr * op

type stmt =
        Block of stmt list
        | Return of expr
        | If of expr * stmt
        | While of expr * stmt
        | Output of term * expr
        | Assign of var * expr
        | SetAIndex of term * term * expr

type func_decl = {
    fname : string;
    params : var list;
    locals : var list;
    body : stmt list;
  }

type program = string list * func_decl list


(* pretty print the ast *)

let string_of_var v =
        (match v.varType with
        Int -> "int "
        | Str -> "str "
        | Arr -> "str[] "
        | Any -> "") ^ v.varName

let rec string_of_term = function
    Integer(i) -> string_of_int i
  | String(s) -> "\"" ^ s ^ "\""
  | Id(s) -> s
  | Array(el) ->
      "[" ^ String.concat ", " (List.map string_of_term el) ^ "]"
  | Null -> ""

let rec string_of_expr = function
   Term(a) -> string_of_term a
  | Call(f, el) -> f ^ "(" ^ String.concat ", " (List.map string_of_expr el) ^ ")"
  | Range(id, a1, a2) -> string_of_term id ^ "[" ^ string_of_term a1 ^ ":" ^ string_of_term a2 ^ "]"
  | Binop(e1, o, a) -> (match o with
        Add -> string_of_expr e1 ^ " + " ^ string_of_term a
        | Sub -> string_of_expr e1 ^ " - " ^ string_of_term a
     | Eq -> string_of_expr e1 ^ " == " ^ string_of_term a
     | Neq -> string_of_expr e1 ^ " != " ^ string_of_term a
     | Lt -> string_of_expr e1 ^ " < " ^ string_of_term a
     | Leq -> string_of_expr e1 ^ " <= " ^ string_of_term a
     | Gt -> string_of_expr e1 ^ " > " ^ string_of_term a
     | Geq -> string_of_expr e1 ^ " >= " ^ string_of_term a
     | And -> string_of_expr e1 ^ " && " ^ string_of_term a
     | Or -> string_of_expr e1 ^ " || " ^ string_of_term a
     | Vat -> string_of_expr e1 ^ "[" ^ string_of_term a ^ "]"
     | _ -> "") (* will never happen, but satisfies compiler *)
  | Unop(v, o) -> (match o with
        Len -> "|" ^ string_of_expr v ^ "|"
        | Not -> "!" ^ string_of_expr v
        | In -> "@" ^ string_of_expr v
     | _ -> "") (* will never happen, but satisfies compiler *)

let rec string_of_stmt = function
    Block(stmts) -> print_endline "block";
      "\n\t" ^ String.concat "\t" (List.map string_of_stmt stmts) ^ "\n"
  | Return(expr) -> "RETURN " ^ string_of_expr expr ^ "\n";
  | If(e, s) -> "IF (" ^ string_of_expr e ^ ")\n" ^ string_of_stmt s
  | While(e, s) -> "WHILE (" ^ string_of_expr e ^ ") " ^ string_of_stmt s
  | Output(dest, src) -> "@" ^ string_of_term dest ^ " = " ^ string_of_expr src
  | Assign(v, e) -> string_of_var v ^ " = " ^ string_of_expr e
  | SetAIndex(id, i, e) -> string_of_term id ^ "[" ^ string_of_term i ^ "] = " ^ string_of_expr e

let string_of_fdecl fdecl =
  fdecl.fname ^ "(" ^ String.concat ", " (List.map string_of_var fdecl.params) ^ ")\n\t" ^
 "Locals: " ^ String.concat ", " (List.map string_of_var (fdecl.locals)) ^ "\n\t" ^
  String.concat "\n\t" (List.map string_of_stmt fdecl.body) ^ "\n"

let string_of_program program =
  String.concat "\n" (List.map string_of_fdecl (List.rev (snd program)))
```

8.4 Bytecode.ml

```
type bstmt =
    Int of int    (* add an int to the heap *)
  | Str of string (* add a string to the heap *)
  | Arr of int * int   (* array length, is empty *)
  | Drp           (* Discard a value *)
  | Opr of Ast.op * Ast.typ * int (* Perform all operations *)
  | Lod of int    (* Fetch local variable *)
  | Sto of int    (* Store local variable *)
  | Jsr of int    (* Call function by absolute address *)
  | Ent of int    (* Push FP, FP -> SP, SP += i *)
  | Ret 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 = {
    text : bstmt array; (* Code for all the functions *)
  }

let string_of_type = function
       Ast.Int -> "I"
       | Ast.Arr -> "A"
       | Ast.Str -> "S"
       | Ast.Any -> "NE"

let string_of_stmt = function
    Int(i) -> "Int " ^ string_of_int i
  | Str(s) ->  "Str \"" ^ s ^ "\""
  | Arr(l, f) -> "Arr " ^ string_of_int l ^ " " ^ string_of_int f
  | Drp -> "Drp"
  | Opr(Ast.Add, t, a) -> "Add " ^ string_of_type t ^ " " ^ string_of_int a
  | Opr(Ast.Sub, t, a) -> "Sub " ^ string_of_type t ^ " " ^ string_of_int a
  | Opr(Ast.Eq, t, a) -> "Eql " ^ string_of_type t ^ " " ^ string_of_int a
  | Opr(Ast.Neq, t, a) -> "Neq " ^ string_of_type t ^ " " ^ string_of_int a
  | Opr(Ast.Lt, t, a) -> "Lt " ^ string_of_type t ^ " " ^ string_of_int a
  | Opr(Ast.Leq, t, a) -> "Leq " ^ string_of_type t ^ " " ^ string_of_int a
  | Opr(Ast.Gt, t, a) -> "Gt " ^ string_of_type t ^ " " ^ string_of_int a
  | Opr(Ast.Geq, t, a) -> "Geq " ^ string_of_type t ^ " " ^ string_of_int a
  | Opr(Ast.And, t, a) -> "And " ^ string_of_type t ^ " " ^ string_of_int a
  | Opr(Ast.Or, t, a) -> "Or " ^ string_of_type t ^ " " ^ string_of_int a
  | Opr(Ast.Vat, t, a) -> "Vat " ^ string_of_type t ^ " " ^ string_of_int a
  | Opr(Ast.Not, t, a) -> "Not " ^ string_of_type t ^ " " ^ string_of_int a
  | Opr(Ast.In, t, a) -> "In " ^ string_of_type t ^ " " ^ string_of_int a
  | Opr(Ast.Len, t, a) -> "Len " ^ string_of_type t ^ " " ^ string_of_int a
  | Opr(Ast.Out, t, a) -> "Out " ^ string_of_type t ^ " " ^ string_of_int a
  | Opr(Ast.Rng, t, a) -> "Rng " ^ string_of_type t ^ " " ^ string_of_int a
  | Opr(Ast.Sai, t, a) -> "Sai " ^ string_of_type t ^ " " ^ string_of_int a
  | Lod(i) -> "Lod " ^ string_of_int i
  | Sto(i) -> "Sto " ^ string_of_int i
  | Jsr(i) -> "Jsr " ^ string_of_int i
  | Ent(i) -> "Ent " ^ string_of_int i
  | Ret(i) -> "Ret " ^ 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 =
  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 *)
    local_index    : int StringMap.t; (* FP offset for args, locals *)
  }

(* val enum : int -> 'a list -> (int * 'a) list *)
(* takes a normal list and rebuilds it with stack indices as fst of pair *)
let rec enum stride n = function
    [] -> []
  | hd::tl -> (n, hd) :: enum stride (n+stride) tl

(* get only the names of vars from var types *)
```

```
let rec get_names vars = List.map (fun v -> v.varName) 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 (others, functions) =

  (* Assign indexes to function names *)
  let function_indexes = string_map_pairs StringMap.empty
      (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_params = List.length fdecl.params
    and num_locals = List.length fdecl.locals
    and local_offsets = enum 1 1 (get_names fdecl.locals)
    and param_offsets = enum (-1) (-2) (get_names fdecl.params) in
    let env = { env with local_index = string_map_pairs
                 StringMap.empty (local_offsets @ param_offsets) } in
(*

Define ast members

*)
let typ = function
       Integer(i)-> Ast.Int
       | String(s) -> Ast.Str
       | Array(a) -> Ast.Arr
       | Id(i) -> Ast.Any
       | Null -> Ast.Any

in let rec term = function
       | Id id -> (try [Lod (StringMap.find id env.local_index)]
              with Not_found -> raise (Failure ("cannot reference undeclared variable " ^ id)))
       | Integer i -> [Int i]
       | String s -> [Str s]
       | Array a -> let l = List.length a in
                              if (l>0) then (
                                     let f = (if typ (List.hd a) = Ast.Str then 1 else 0) in
                                     (List.concat (List.map term a)) @ [Arr ((l), f)])
                              else [Arr (0, 0)]
       | Null -> []

in let rec expr = function
       Term (a) -> term a
       | Call (fname, args) -> (try
         (List.concat (List.map expr (List.rev args))) @
         [Jsr (StringMap.find fname env.function_index) ]
        with Not_found -> raise (Failure ("undefined function " ^ fname)))
       | Range (id, a1, a2) -> term a1 @ term a2 @ term id @ [Opr(Ast.Rng, typ a2, 3)]  (* Opr op typ args *)
       | Binop (e, op, a) -> expr e @ term a @ [Opr (op, (typ a), 2)]
       | Unop (e, op) -> expr e @ [Opr (op, Ast.Int,1)]

in let rec stmt = function
       Block sl     ->  List.concat (List.map stmt sl)
       | Return e     -> expr e @ [Ret num_params]
       | If (p, t) -> let t' = stmt t in expr p @ [Beq(1 + List.length t')] @ t'
    | 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'))]
        | Assign (s, e) -> expr e @ [Sto (StringMap.find s.varName env.local_index)]
        | SetAIndex(id, i, e) -> expr e @ term i @ term id @ [Opr(Ast.Sai, Ast.Arr , 3)]
        | Output(t, e) -> expr e @ [Opr (Ast.Out, (typ t), 1)]

    in [Ent num_locals] @      (* Entry: allocate space for locals *)
    stmt (Block fdecl.body) @  (* Body *)
    [Int 0; Ret num_params]   (* Default = return 0 *)

  in let env = { function_index = function_indexes;
                 local_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

  {
    (* 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 HC
   Saved FP   <-- FP
   Saved PC
   Arg 0
   ...
   Arg n *)

let execute_prog prog d =
  let stack = Array.make 1024 0
  and heap = Array.make 1024 "" in

(** maintenance tools *)
(* print debug statements *)
let debug fp sp pc hc =
        print_string "stack = "; Array.iteri (fun index value ->
                if (index < sp) then print_string
                        (string_of_int index ^ ":" ^ string_of_int value ^ " ")) stack;
        print_string "\nheap = "; Array.iteri (fun index value ->
                if (index < hc) then print_string
                        (string_of_int index ^ ":" ^ value ^ " ")) heap;
        print_endline "\n----------------------";
        print_endline ((string_of_int pc) ^ ": " ^
               Bytecode.string_of_stmt prog.text.(pc) ^
               " (fp:" ^  (string_of_int fp) ^
               "; sp:" ^ (string_of_int sp) ^
               "; hc:" ^ (string_of_int hc) ^ ") -->")

(** execution helper functions *)

(* add string to heap; return a heap pointer to that string *)
in let set_S hc str =
       heap.(hc) <- str; hc
(* get an int pointer to the string on the heap *)
in let get_S key = if key = -1 then ""
       else heap.(key)


(* allocate a new array on the heap *)
in let new_A hc l f =
       let t = (l>0) in match (t,f) with
       (* array with pointers to its strings immediately before it on heap *)
       (true, 1) -> let rec a = function
                      0 -> ""
                      | len -> (string_of_int (hc-len)) ^ ";" ^ (a (len-1))
                      in set_S hc (a l)
       (* alloc an array of size l with null pointers *)
       |(true, 0) -> let rec a = function
                      0 -> ""
                      | len -> (string_of_int (-1)) ^ ";" ^ (a (len-1))
                      in set_S hc (a l)
       (* empty array pointer *)
       |(false, 0) -> set_S hc ""
       | _ -> hc
(* save an array's string on the heap *)
in let set_A_ref hc a =
       set_S hc (String.concat ";" (List.map string_of_int (Array.to_list a)))
(* returns int list with pointers to strings on heap *)
(* string Array -> int -> int list *)
in let get_A_ref key =
       let a_ref_strList = Str.split (Str.regexp ";") heap.(key) in
       List.map int_of_string a_ref_strList
(* list of pointer ints -> list of actual strings *)
in let get_A key =
       let a = get_A_ref key
       and value k = match k with -1 -> "" | _ -> (heap.(k)) in
       List.map value a

(* set int values for bools *)
in let boolean i = if i then 1 else 0
in let booli i = if (i=1) then true else false
in let nbooli i = if (i=0) then true else false

(** enter program execution *)
in let rec exec fp sp pc hc =
  if (d = 1) then debug fp sp pc hc ;
  match prog.text.(pc) with
    Int i  -> stack.(sp) <- i ; exec fp (sp+1) (pc+1) hc
  (* add a string to the heap *)
  | Str s       -> stack.(sp) <- set_S hc s; exec fp (sp+1) (pc+1) (hc+1)
  (* add an array ref indicating length to the heap *)
  | Arr (l, f)  -> stack.(sp) <- new_A hc l f; exec fp (sp+1) (pc+1) (hc+1)
  | Drp    -> exec fp (sp-1) (pc+1) hc (* Discard a value *)
  | Opr (op, typ, args) -> (match (typ, args) with
       (* Rng of string *)
       (Ast.Str,3) ->  let b = stack.(sp-2) and n = stack.(sp-1)
                              and s = get_S stack.(sp-3) in
                              let f = (Str.string_after (Str.string_before s (n+1)) b) in
                              stack.(sp-3) <- set_S hc f; exec fp (sp-2) (pc+1) (hc+1)
       |(Ast.Arr,3) -> (match op with
               Sai            -> let e = stack.(sp-3) and i = stack.(sp-2)
                                     and a = Array.of_list (get_A_ref stack.(sp-1)) in (* yields int array *)
                                     stack.(sp-3) <- set_S a.(i) (get_S e); exec fp (sp-2) (pc+1) (hc+1)
               | Rng   -> let b = stack.(sp-2) and n = stack.(sp-1)
                                     and a = get_A_ref stack.(sp-3) in
                                     let f = Array.sub (Array.of_list a) b (n-b+1) in
                                     stack.(sp-3) <- set_A_ref hc f; exec fp (sp-2) (pc+1) (hc+1)
                 |_             -> exec fp sp (pc+1) hc)
       (* binops for stack arithmetic/boolean eval of ints *)
       |(Ast.Int,2) -> let op1 = stack.(sp-2) and op2 = stack.(sp-1) in
       stack.(sp-2) <- (match op with
               Add            -> op1 + op2
       | Sub     -> op1 - op2
       | Eq   -> boolean (op1 = op2)
       | Neq     -> boolean (op1 != op2)
       | Lt           -> boolean (op1 < op2)
       | Leq     -> boolean (op1 <= op2)
       | Gt   -> boolean (op1 > op2)
       | Geq     -> boolean (op1 >= op2)
       | And  -> boolean ((booli(op1) && booli(op2)))
       | Or           -> boolean ((booli(op1) || booli(op2))));
       exec fp (sp-1) (pc+1) hc
       |(Ast.Str,2) -> let op1 = get_S(stack.(sp-2)) in (match op with
            Vat               -> let op2 = stack.(sp-1) in
                      stack.(sp-2) <- set_S hc (String.make 1 op1.[op2]); exec fp (sp-1) (pc+1) (hc+1)
       | Add  -> let op2 = get_S(stack.(sp-1)) in
               stack.(sp-2) <- set_S hc (op1 ^ op2); exec fp (sp-1) (pc+1) (hc+1)
       | Eq   -> let op2 = get_S stack.(sp-1) in
               stack.(sp-2) <- boolean (nbooli(String.compare op1 op2)); exec fp (sp-1) (pc+1) (hc)
       | Neq     -> let op2 = get_S stack.(sp-1) in
               stack.(sp-2) <- boolean (not(nbooli(String.compare op1 op2))); exec fp (sp-1) (pc+1) (hc))
       |(Ast.Any,1) -> let msg = get_S stack.(sp-1) in print_endline msg; exec fp (sp) (pc+1) (hc)

       |(Ast.Arr,2) -> (match op with
               Vat     -> let i = stack.(sp-2) and a = Array.of_list (get_A_ref stack.(sp-1)) in
                              stack.(sp-2) <- a.(i); exec fp (sp-1) (pc+1) (hc)
       | Add -> let a1 = (get_A_ref stack.(sp-2)) and a2 = (get_A_ref stack.(sp-1)) in
                              stack.(sp-2) <- set_A_ref hc (Array.of_list(a1 @ a2)); exec fp (sp-1) (pc+1)
(hc+1)
       | Eq  -> let a1 = (get_A stack.(sp-2)) and a2 = (get_A stack.(sp-1)) in
                              stack.(sp-2) <- boolean(a1 = a2); exec fp (sp-1) (pc+1) (hc)
       | Neq -> let a1 = (get_A stack.(sp-2)) and a2 = (get_A stack.(sp-1)) in
                              stack.(sp-2) <- boolean(a1 <> a2); exec fp (sp-1) (pc+1) (hc))
       |(Ast.Int ,1) -> let op1 = stack.(sp-1) in (match op with
               Not     -> stack.(sp-1) <- boolean(nbooli(op1));
               exec fp (sp) (pc+1) (hc))
       |(Ast.Str ,1) -> let op1 = get_S stack.(sp-1) in    (match op with
(** TBD        In  -> let input = "1"  in
               stack.(sp-1) <- set_S hc input; exec fp (sp-1) (pc+1) (hc+1)
               let filename = "test" in if Sys.file_exists then
               else raise(Failure "Input file (" ^ filename ^ ") not found")  *)

               | Len -> stack.(sp-1) <- String.length op1; exec fp (sp) (pc+1) (hc))
       |(Ast.Arr ,1) -> (match op with
               | Len -> stack.(sp-1) <- List.length (get_A_ref stack.(sp-1)); exec fp (sp-1) (pc+1) (hc)))

  | Lod i   -> stack.(sp)   <- stack.(fp+i) ; exec fp (sp+1) (pc+1) hc
  | Sto i   -> stack.(fp+i) <- stack.(sp-1) ; exec fp sp (pc+1) hc
  | Jsr i   -> stack.(sp)   <- pc + 1       ; exec fp (sp+1) i hc
  | Ent i   -> stack.(sp)   <- fp; stack.(sp+1) <-hc; exec (sp) (sp+i+2) (pc+1) hc
  | Ret i   -> let new_fp = stack.(fp) and new_sp = stack.(fp-i-1)
                      and new_pc = stack.(fp-1) and new_hc = stack.(fp+1) in
                      stack.(fp-i) <- stack.(sp-1) ;
                      exec new_fp new_sp new_pc new_hc
  | Beq i   -> exec fp (sp-1) (pc + if stack.(sp-1) =  0 then i else 1) hc
  | Bne i   -> exec fp (sp-1) (pc + if stack.(sp-1) != 0 then i else 1) hc
  | Bra i   -> exec fp sp (pc+i) hc
  | Hlt     -> ()

  in exec 0 0 0 0
```

8.7 scam.ml

```
type action = Ast | Invalid | Bytecode | Compile

let _ =

if Sys.argv.(1) = "-h" then
  print_endline "help is on the way!" (* do nothing. *)
else
  let action = if Array.length Sys.argv > 1 then
    List.assoc Sys.argv.(1) [ ("-a", Ast);
                             ("-b", Bytecode);
                            ("-c", Compile)]
  else Compile in
  let debug = if (Array.length Sys.argv > 2) && (Sys.argv.(2) = "-d") then 1 else 0 in
  let lexbuf = Lexing.from_channel stdin in
  let program = Parser.program Scanner.token lexbuf in
 (* let program = Semantics.check program 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) debug
```

8.8 Semantics.mlx (does not compile, but hey I tried)

```
open Ast

let stab = ref []

let print_stab  s = print_endline "stab = ";
       List.iter (fun v -> print_string (Ast.string_of_var v)) s

let typestring = function
        Ast.Int-> "INT"
       | Ast.Str -> "STR"
       | Ast.Arr -> "ARR"
       | Ast.Any -> "ANY"

let rec term = function
       Integer i -> Ast.Int
       | String(s) -> Ast.Str
       | Array(a) -> Ast.Arr
       (* check the stab for a var and get its explicit or dynamic type  *)
       | Id(i) -> let v = try (List.find (fun x -> x.varName = i) !stab)
                                             with Not_found-> raise (Failure ("Undeclared variable"))
                      in if (Ast.Any <> v.varType) then  v.varType
                      (*else(if (v.dType <> Ast.Any) then v.dType *)
       | Null -> Ast.Any

let rec expr = function
       Term (a) -> term a
       (** bound to cause problems *)
       | Call (fname, args) -> Ast.Any
       | Range (id, a1, a2) -> let t1 = term a1 and t2 = term a2 and i = term id in
              if ((t1 = Ast.Int) && (t2 = Ast.Int) && (i <> Ast.Int)) then i else
              raise (Failure ("Invalid sub-string/sub-array expression"))
       | Binop(e, o, a) -> let op1 = expr e and op2 = term a in
                      let eq = if (op1 = op2) then op2 else raise (Failure
                       ("A binary operation was attempted on incompatible types"))
                       in let ints i = if (i = Ast.Int) then i else raise (Failure
                       ("An arithmetic/boolean operation was attempted on non-integers"))
                       in (match o with
               Vat -> let m = expr e in

                       if (m <> Ast.Int )
                       then (
                               let i = term a in  (
                                       if (i = Ast.Int)
                                       then i
                                       else raise (Failure ("Invalid string/array reference index supplied"))
                               )
                       )
                       else raise (Failure    ("Invalid string/array"))

               | Lt -> ints eq
               | Leq -> ints eq
               | Gt -> ints eq
               | Geq -> ints eq
               | And -> ints eq
               | Or -> ints eq
               | Sub ->       ints eq
               | _ -> eq)
       | Unop(v, o) -> (match o with
               Len -> let e = expr v in if(e <> Ast.Int) then e  else raise (Failure
                       ("Cannot compute the length of this argument"))
               | Not -> let e = expr v in if(e = Ast.Int) then e  else raise (Failure
                       ("An boolean operation was attempted on non-integers"))
               | In -> let e = expr v in if(e <> Ast.Arr) then e  else raise (Failure
                       ("Invalid input source")))

let rec stmt = function
    Block(stmts) -> List.iter stmt stmts
  | Return(e) -> expr e; ()
  | If(e, s) -> let e1 = expr e in if(e1 = Ast.Int) then stmt s  else raise (Failure
                       ("Invalid predicate in If statement")); ()
  | While(e, s) -> let e1 = expr e in if(e1 = Ast.Int) then stmt s  else raise (Failure
                       ("Invalid predicate in While statement")) ; ()
  | SetAIndex(id, i, e) -> let t1 = term id and t2= term i and e1 = expr e in
       if((t1 = Ast.Arr) && (t2 = Ast.Int) && (e1 <> Ast.Int)) then e1
       else raise (Failure ("Invalid array index assignment"))  ; ()
  | Output(t, e) -> Ast.Any ;()
  (* set the vartype during assignment if it was not initialized *)
  (* if the var has an explicit type, make sure the expr has that type *)
  | Assign(v, e) -> let eT = expr e in (if (v.varType = Ast.Any) then (v.dType <- eT)
                       else (if (v.varType <> eT)
                               then raise
                               (Failure ("Invalid assignment: " ^ Ast.string_of_var v ^ " to " ^ typestring eT))
                       ));
                       stab := v::!stab; Ast.Any; ()

(*
let type_check func =
       {fname = func.fname;
        params = func.params;
        body = List.map (fun s -> stab:=[]; stmt s) func.body;
        locals = func.locals}
*)
let check (others, functions) =
       List.map (fun s -> stab:=[]; stmt s) functions.body;
       (*let valid_typed_funs = List.map type_check functions in (others, valid_typed_funs); *)
       (others, functions);
```