Trix Final Report:
Matrix Manipulation Language

Andrew S. Hunter
UNI: ash2188

Columbia University
Programming Languages and Translators
August 14, 2015

Table of Contents

1 Introduction

Trix was first intended to be an extensive matrix manipulation language with a library of built in functions and matrix constructors. After many struggles in the development phase, the result is a much simpler language used to perform arithmetic languages. Trix is an imperative language that performs arithmetic operations on integers. It uses many of the traditional arithmetic and conditional operators as defined in this report. There is the ability to define functions and define control flow operations such as if-then-else structures, for loops and while loops.

2 Lessons Learned

The biggest takeaway from this project and the course is that the best approach (for someone with no compiler experience) is to start development with the simplest feature and work incrementally while testing along the way. I started this project with a grand vision of what the compiler would do and I foolishly tried to construct the entire puzzle at once, however it would have been more efficient to attack a very simple subset of the features and expand after each component was finished and tested. By trying to develop every expected feature from the start, I created a "compiler" that did nothing right, which resulted in me having to work backwards, defeaturing and collecting the broken pieces.

In addition to having a poor overall approach, I found that my "planning" of the language/compiler itself was not rigorous enough. I developed a concept for the language with some very high level investigation of the pieces necessary to execute the tasks that I wanted. Once I defined the pieces I assumed that I would be able make it work. This was not enough. I should have done more work up front to investigate how each feature would be implemented from the front-end all the way to the execution of the target language.

Lastly, I should have targeted a language that I was more familiar with. Originally I chose C because I wanted to brush up on my knowledge of the language and thought this project was a good opportunity to do so. Well into the project I found my background knowledge of the language inhibited my ability to do the work. I switched to C++, which I am slightly more familiar with.

Unfortunately, I was never able to get to functioning compiler. The following sections of this report describe the Trix language and compiler. Because I was unable to execute any tests, these are the perceived features of Trix, many of which were wiped from the code due to last-ditch efforts to get something working.

3 Tutorial

3.1 Compile Process

The Trix executable is built with the Makefile procedure. Additionally, there are five source files that are necessary to build. The source code architecture is discussed later. When the executable is created, a program can be compiled using the following.

```
./Trix program_name.trx
```

To compile the generated C++ code, the g++ command of the GCC compiler is used. The following is an example, generating the default a.out.

```
g++ program_name.cpp
```

3.2 Variables

Variable declarations and initializations are performed first with declaration initialization of the type followed by the definition.

```
int new_var;
new_var = 2;
```

3.3 Control Flow

All programs start with the main function. Here's an example of the if-else structure.

```
function main () {

   int test_var;
   test_var = 5;

   if ( test_var == 5 ) {
        int equiv;
        equiv = 1 * test_var;
   }
   else { equiv = 0 }

   return equiv;
}
```

3.4 Functions

Functions are declared with the function keyword. Here is an example of a function definition.

```
function main () {

     function int multiply ( int arg1 , int arg2 ) {
          int result;
          result = arg1 * arg2;
          return result;
     }
}
```

4 Language Reference Manual

4.1 Syntax

The syntax of Trix is defined by the tokens, which are the basic building blocks of a program written in the language. The tokens are defined in the scanner, and anything that does not appear as a token or whitespace is rejected and an exception is thrown. Separation, control flow, and blocks are delineated with parenthesis or braces. The semicolon is also used to identify the end of a statement.

`()`	Parenthesis are used to identify the arguments of a function or identify the passed variables or literals to a function. They are also used to define the conditionals used in the if-else structure, for loop and while loop.
`{}`	Braces are used to delineate the body of a function as well as separate the statements the statements of the if-else structure, for loop and while loop.
`;`	Semicolon used to end a statement and proceed to the next line or statement

4.2 Comments

The comment syntax uses two percent (%%) symbols at the start and finish of a comment. All characters that appear between the comment symbols will be ignored.

```
%% This is a comment %%
```

4.3 Types

Integers are the only primitive type in Trix. The keyword “int” is used to declare and initialize variables and identify return types of functions. Integers must be declared before they are initialized.

declare: `int ID;`
initialize: `ID = 15;`

4.4 Variable Names

Variable names are constructed from a combination of letters and numbers. The variable name must start with a lowercase alphabetical character. Certain keywords are reserved for language specific operations.

4.5 Keywords

The keywords in the Trix language are listed below. The characters are reserved and are used by the compiler to indicate specific characteristics of the language.

```
int
function
if
else
for
while
```

4.6 Arithmetic Expressions

The following outlines the rule for an arithmetic expression. The allowable substitutions are variable names, integer literals, binary operation with two expressions, assignment and function call.

`expr + expr`	**Addition of two expr**
`expr - expr`	**Subtraction of two expr**
`expr * expr`	**Multiplication of two expr**
`expr / expr`	**Division of two expr**

4.7 Conditional Expressions (Removed)

4.8 Control Flow

The if-else control structure is defined with the following syntax. The statements are executed based on the conditional expression. The expression returns a boolean true or false, and if it evaluates true, the statement is executed. Note: the conditional expressions were removed and therefor the control flows below were not expected to execute.

```
if (expression)
```

```
        {statement;}
else
        {statement;}
```

The while loop is defined with the following syntax. Like the if-else, it requires parenthesis for the conditional statement.

```
while (expression) {
     {statement;}
   }
```

4.9 Functions

Functions are defined with the following syntax. When a function returns a value, the type of the value must be defined. Following the return type is the name of the function and the arguments, which are wrapped with parenthesis. The body of the function follows, and is wrapped with braces.

```
function int name ( args ) {
        statements;
}
```

4.10 Scope

Variables that are defined within the global scope (main) are accessible by all functions or blocks within the program. Variables that are defined within a function are only accessible to the function.

5 Architectural Design

The Trix compiler is separated into four distinct phases: scanning, parsing, translating, and compiling. A semantic/type checking phase was not included due to the simplicity of the Trix language.

5.1 Scanning

The scanning phase is executed utilizing the token definitions in scanner.mll. The scanner takes the source code and produces a group of tokens. The tokens are the relevant building blocks of the language.

5.2 Parsing

The parsing phase takes the token group from the scanner and the abstract syntax tree structure defined in ast.ml and produces an abstract syntax tree for the program being compiled. This phase defines the structure of the language.

5.2 Translation

The translation phase is performed utilizing the file target.ml file. This is the phase where the compiler steps through the abstract syntax tree produced by the parsing phase and generates C++ code based on the rules defined in target.ml. The trix.ml file is used open and write the code generated from the target.ml file.

5.3 Compile

Once the source program has been translated to the target language and a text file is generated, the program can be compiled with the target compiler. In this case a GCC compiler is used to compile C++ code.

6 Source Code

scanner.mll

```
(* Andy Hunter
PLT Summer 15
Trix *)

{ open Parser }

rule token = parse
  | [' ' '\t' '\r' '\n'] { token lexbuf }
  | "%%" { comment lexbuf }
  | "int" { INT }
  | "if" { IF }
  | "else" { ELSE }
  | "while" { WHILE }
  | "return" { RETURN }
  | "function" { FUNCTION }
  | '+' { PLUS }
  | '-' { MINUS }
  | '*' { TIMES }
  | '/' { DIVIDE }
  | '=' { ASSIGN }
  | '(' { LPAREN }
  | ')' { RPAREN }
  | '{' { LBRACE }
  | '}' { RBRACE }
  | ',' { COMMA }
  | ';' { SEMI }
  | ['0'-'9']+    as lxm { INT_LIT (int_of_string lxm) }
  | ['a'-'z' 'A'-'Z']['a'-'z' 'A'-'Z' '0'-'9' '_']* as lxm { ID(lxm) }
  | eof { EOF }
  | _ as char { raise (Failure("illegal character " ^ Char.escaped char)) }

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

parser.mly

```
/* Andy Hunter
PLT Summer 15
Trix */

%{ open Ast %}

%token SEMI LPAREN RPAREN LBRACE RBRACE COMMA
%token PLUS MINUS TIMES DIVIDE ASSIGN
%token RETURN IF ELSE WHILE INT
%token FUNCTION
%token <int> INT_LIT
%token <string> ID
```

```
%token EOF

%right ASSIGN
%left PLUS MINUS
%left TIMES DIVIDE
%nonassoc ELSE

%start program
%type <Ast.program> program

%%

datatype:
  | INT { Int }

expr:
  | ID { Id($1) }
  | INT_LIT { IntLiteral($1) }
  | 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) }
  | ID ASSIGN expr { Assign($1, $3) }
  | ID LPAREN passed_args RPAREN { ApplyFunct($1, $3) }

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

stmt:
  | expr SEMI { Expr($1) }
  | RETURN expr SEMI { Return($2) }
  | LBRACE stmt_list RBRACE { Block(List.rev $2) }
  | IF LPAREN expr RPAREN stmt ELSE stmt { If($3, $5, $7) }
  | WHILE LPAREN expr RPAREN stmt { While($3, $5) }

stmt_list:
  | { [] }
  | stmt_list stmt  { $2 :: $1 }

vardef:
  | vardecl { VarDecl($1) }
  | varinit { Varinit($1) }

vardef_list:
  | { [] }
  | vardef_list vardef { $2 :: $1 }

vardecl:
  | datatype ID SEMI { {varname=$2;vartype=$1} }

varinit:
  | datatype ID ASSIGN expr SEMI { Vinit ( {varname = $2; vartype = $1} , $4) }

functdecl:
```

```
  | FUNCTION datatype ID LPAREN arg_list RPAREN LBRACE vardef_list stmt_list RBRACE
    { { ret = $2; name = $3; args = $5; locvars = $8; body = $9;} }

functdef:
  | datatype ID { {varname=$2; vartype = $1} }

arg_list:
  | { [] }
  | functdef              { [$1] }
  | arg_list COMMA functdef  { $3 :: $1 }

program:
  | { [], [] }
  | program vardef { ($2 :: fst $1), snd $1 }
  | program functdecl { fst $1, ($2 :: snd $1) }
```

ast.ml

```
(* Andy Hunter
PLT Summer 15
Trix *)

type op = Add | Sub | Mult | Div

type dtype = Int

type var_decl = {varname: string; vartype: dtype}

type expr =
  | Id of string
  | IntLiteral of int
  | Binop of expr * op * expr
  | Assign of string * expr
  | ApplyFunct of string * expr list

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

type var_init =
  | Vinit of var_decl * expr

type var_def =
  | Varinit of var_init
  | VarDecl of var_decl

type func_decl = {
    name : string;
    args : var_decl list;
    locvars : var_def list;
    body : stmt list;
```

```
   ret: dtype
 }

type program = var_def list * func_decl list
```

Target.ml

```
(* Andy Hunter
PLT Summer 15
Trix *)

open Ast

(* string helpers *)
let vartype_str = function
  | Int -> "int"

let rec expr_str = function
  | Id(i) -> i
  | IntLiteral(j) -> string_of_int j
  | Binop(k1, d, k2) -> expr_str k1 ^ " " ^ (match d with Add -> "+" | Sub -> "-" | Mult -> "*" | Div -> "/")
^ " " ^ expr_str k2
  | ApplyFunct(m, n) -> m ^ "(" ^ String.concat ", " (List.map expr_str n) ^ ")"
  | Assign(a, b) -> a ^ " = " ^ expr_str b

let rec stmt_str = function
  | Expr(expr) -> expr_str expr ^ ";"
  | Return(expr) -> "return " ^ expr_str expr ^ ";";
  | If(a, b1, b2) ->  "if (" ^ expr_str a ^ ") {" ^ stmt_str b1 ^ "} else {" ^ stmt_str b2 ^ "}"
  | While(i, j) -> "while (" ^ expr_str i ^ ") {" ^ stmt_str j ^ "}"
  | Block(stmt) -> "{" ^ String.concat "" (List.map stmt_str stmt) ^ "}"

let vardecl_str id = (vartype_str id.vartype) ^ " " ^ id.varname

let varinit_str = function
 | Vinit(a, b) -> vardecl_str a ^ " = " ^ expr_str b

let vardef_str = function
 | VarDecl(a) -> vardecl_str a ^ ";"
 | Varinit(b) -> varinit_str b ^ ";"

let functdecl_str functdecl =
  functdecl.name ^ " (" ^ String.concat ", " (List.map vardecl_str functdecl.args) ^ ") {" ^
  (String.concat " " (List.map vardef_str functdecl.locvars)) ^
  String.concat " " (List.map stmt_str functdecl.body) ^ "}"

let prog_str (a, b) =
  String.concat " " (List.map vardef_str a) ^ " " ^ String.concat " " (List.map functdecl_str b)

module StringMap = Map.Make(String)

type scope = { functsig: string StringMap.t; globals:  dtype StringMap.t; locals:  dtype StringMap.t;}

let funct_string functdecl =
```

```
   functdecl.name ^ "_" ^ String.concat "_" (List.map vardecl_str functdecl.args)

let vinitial = function
   | Vinit(c, _) -> (c.vartype, c.varname)

let getvnamedef = function
   | VarDecl (a) -> (a.vartype, a.varname)
   | Varinit(b) -> vinitial b

let rec svardef = function
   | [] -> []
   | hd::tl -> ( getvnamedef hd) :: svardef tl

let rec getvardef = function
   | [] -> []
   | hd::tl -> (hd.vartype, hd.varname) :: getvardef tl

let rec sfunct = function
   | [] -> []
   | hd::tl -> (funct_string hd, hd.name) ::  sfunct tl

let mapnew x y =
  List.fold_left (fun a (b, c) -> StringMap.add c b a) x y

let trix_target (globals, functions) out =

  let global_Vars = mapnew StringMap.empty (svardef globals) in
  let functsigs = mapnew StringMap.empty (sfunct functions) in

  let targetnew scope functdecl =
    let locals = svardef functdecl.locvars
    and argvars = getvardef functdecl.args in
    let scope = mapnew StringMap.empty (locals @ argvars) in

    let rec expr a =
    (match a with
      | IntLiteral(b) ->  string_of_int b
      | Id(c) -> if ((StringMap.mem c scope.locals) || (StringMap.mem c scope.globals)) then c
        else raise (Failure ("no id:  " ^ c ))
     | Binop(d1,f,d2) ->  expr d1 ^ " " ^ (match f with Add -> "+" | Sub -> "-" | Mult -> "*" | Div -> "/") ^ "
" ^ expr d2
     | Assign(v, a) ->  if ((StringMap.mem v scope.locals) || (StringMap.mem v scope.globals)) then v ^
"=" ^ expr a
            else raise (Failure ("no id: " ^ v))
     | ApplyFunct(s, e) ->  if (StringMap.mem s scope.functsig) then ( s ^ "(" ^ (String.concat ","
(List.map
                        expr (List.rev e))) ^ ");") else raise (Failure ("no function: " ^ s)))

   in let rec stmt = function
     | Expr(a) -> expr a ^ ";"
     | Return(b) -> "return " ^ expr b ^ ";"
     | If(c,d,e)  -> "if(" ^ expr c ^ "){ " ^ stmt d ^ "} else {" ^ stmt e ^ "} "
     | While(e,s) -> "while(" ^ expr e ^ ") {" ^ stmt s ^ " } "
     | Block(stmts) -> String.concat "" (List.map stmt stmts) ^ " "
```

```
  let scope = { functsig = functsigs; globals = global_Vars; locals = StringMap.empty } in

  let main_run = try
    (StringMap.find "main" functsigs) with Not_found -> raise (Failure ("no main"))
  in String.concat " " (List.map vardef_str (List.rev globals)) ^ " " ^
  (String.concat  " " (List.map (targetnew scope) (List.rev functions)))
```

trix.ml

```
(* Andy Hunter
PLT Summer 15
Trix *)

open Printf

let filename = ref "output"

let writefile name line =
  let outputfile = open_out name in
    fprintf outputfile "%s\n" line;
    close_out outputfile

let getfile path =
  let dir = (String.rindex path '/') in
          let len = (String.length path) - (dir + (String.length ".trx" +1)) in
          String.sub path (dir + 1) (len)

let main () =
  let lexbuf = ignore(filename := getfile Sys.argv.(2));
           Lexing.from_channel (open_in Sys.argv.(2)) in
  let program = Parser.program Scanner.token lexbuf in
                let test1 = Target.trix_target program !filename in
          writefile (!filename ^ ".cpp") test1
```