DYNAMO - A Language for Dynamic Programming

Final Report

Raghavan Muthuregunathan - rm2903

Pradeep Dasigi - pd2359

Abhinav Saini - as3906

Archana Balakrishnan - ab3416

Srilekha VK - sv2344

1. Introduction

1.1 Dynamic Programming

According to Wikipedia, "Dynamic Programming (DP) is a method for solving complex problems by breaking them down into simpler subproblems." Dynamic programming problems have the following characteristics.

- Overlapping subproblems

  The main problem is solved by reusing the solutions to the smallest sub-problem and then building up. For example, the factorial of a number is computed by multiplying the number and the factorial of the next (smaller) number, which in turn is computed the same way, and so on. i.e. Factorial(N) = N*Factorial(N-1); Factorial(N-1) = (N-1)* Factorial(N-2) and so on. After the function is executed for a base argument – Factorial(1), it is computed for Factorial(2).

- Optimal substructure

  The optimal solution to the main problem is constructed by optimally solving its sub-problems. For example, in the Dijkstra shortest path algorithm, the shortest path from A (source) to Z (target) is the shortest path from A to B (node from A that is at the least distance from A) added to the shortest path from B to Z, and so on.

In programming languages, DP problems are typically solved by recursive functions.

1.2 Why does DP need a different language?

Writing DP programs in high-level general-purpose languages is complex and requires extensive book-keeping. Added to this complexity is the burden of following the complex syntactic rules of the high-level language. Dynam-o is a writable language for coding DP programs. The programmer codes the solution to the problem in a straight-forward way, by doing just the following.

- Defining operation to be executed for base values – such as Fibonacci(1) and Fibonacci(2) for Fibonacci series, and Factorial(1) for Factorial computation programs.

- Defining operations (recursive operations) to be executed for values other than base values – Fibonacci(3), Fibonacci(4), etc., and Factorial(2), Factorial(3), etc.
- Defining a conditional, in the form of an 'if' statement
- Specifying the recursive operation in the then statement, and the base operation in the else statement.

The goal is to make the writing of the program intuitive – as close as possible to te writing the problem's core mathematical equations.

1.3 Why Java?

Java is one of the most popular programming languages today, and one of its best features is portability. The byte-code output of the Java compiler is completely machine-independent and can run on any architecture.

In Dynam-o, the programmer has the option of including Java code between two '$' symbols (just like comments are specified in Java between /* and */). Java programs may be used to supplement a Dynam-o program with features not currently supported in Dynam-o.

2. Language Tutorial

A program in Dynamo bears the following structure.

```
<ProgramName>
{

        {
                <Input Statements>
        }
        {
                <Logic Statements>
        }

}
```

ProgramName is an identifier in Dynamo, and therefore, programs must be named according to the rules specified for identifiers.

The ‘Input Statements’ define the input variables for which the solution needs to be computed. The ‘Logic Statements’ specify the logic required to solve the DP problem in the form of conditionals, then statements and else statements.

Example:

```
LevDist.dyn

LevDist
{
   {
     data-init str1;
     data-init str2;
   }
   {
      if (str1 = 0 & str2 = 0)
            0;
      else if (str2 = 0)
            #str1;
      else if (str1 = 0)
            #str2;
      else if (str1[i1] = str2[i2])
            LevDist(i1-1, i2-1);
      else
                min(LevDist(i1-1,i2-1)+1,
LevDist(i1,i2-1)+1,LevDist(i1-1,i2)+1);
    }
}
```

```
LevDist.java

public class LevDist {
       public  static  int  LevDist(String
str1, String str2) {
             int temp1 = str1.length()+1;
             int temp2 = str2.length()+1;
             int    temp[][]    =    new
int[temp1][temp2];
             for(int i=0;i<temp1;i++) {
             for(int j=0;j<temp2;j++) {
                    if(i==0&&j==0) {
                           temp[i][j]=0;
                    }
                    else if(j==0) {
                           temp[i][j]=i;
                    }
                    else if(i==0) {
                           temp[i][j]=j;
                    }
                    else
if(str1.charAt(i-1)==str2.charAt(j-1)) {
                           temp[i][j]
=temp[i-1][j-1];
                    }
                    else {
                           int   min   =
temp[i-1][j-1]+2;
                           min         =
min>(temp[i][j-1]+1)?temp[i][j-1]+1:min;
                           min         =
min>(temp[i-1][j]+1)?temp[i-1][j]+1:min;
                           temp[i][j]=min;
                    }
             }
             }
             return  temp[temp1-1][temp2-
1];
       }

       public  static  void  main(String
args[]) {
             int     LevDist_res     =
LevDist(args[0], args[1]);

       System.out.println("\n"+LevDist_res
);
       }
       }
```

3. Revised Language Manual

Lexical Conventions

The tokens are identifiers, keywords and constants. Blank lines, newline characters, spaces, tabs and comments are ignored and are used only to separate tokens.

The Dynamo scanner identifies the following entities as valid:

3.1 Comments
Comments in Dynam-o are bounded by /* at the beginning and */ at the end.

3.2 Embedded Java code
Java code is bound by a '$' at the beginning and a '$' at the end.

3.3 Identifiers
Identifiers are valid strings that are either keywords or names. A name is either a program name or a variable name. Identifiers can contain alphabets, numbers and/or underscore (_) in any order. There is no length imposed.

3.4 Separators
The following separators are used.
Opening and Closing curly braces ('{' and '}') are used to contain the program and the logic statements. They should be specified at the beginning of the program (after the mention of the program name) and the end of a program (after the closing brace of the logic block of statements), and at the beginning and end of the logic block of statements.
Opening and Closing parentheses('(' and ')') are used around conditionals.
Semicolon (';) is used as a statement terminator.

3.5 Keywords:
data-init – used to define the input variables
if – used to check if a value is a base value or a recursive value; following statement is executed if the value is a recursive value.
else – The statement following an 'else if' is executed if the condition checks a recursive value; the statement following an 'else' is executed if all previous conditionals are false.
max - is predefined as DP problems are usually optimization problems
min - predefined similarly like max.

3.6 Data Types
Dynam-o uses only one data-type – String. For problems that require integer inputs, the string input values are converted to integers. This can be done by prefixing the '#' symbol.

3.7 Constants
Constants are either of string data-type or integer data-type. String constants are checked in conditionals. Integer constants are returned by the program.

3.8 Operators and Overloading

Overloaded operators
+ : integer addition, string concatenation
- : integer subtraction
/ : integer division
* : integer multiplication
= : integer | boolean | char | string equality
Logical Operators
& - And
| - Or
= - Logical Equal To
> - Greater than
>= - Greater than or equal to
< - Lesser than
<= Lesser than or equal to
!= - Not equal to
Other operators
# - operator for computing the length of the string (eg. #str returns length of string str)
[and] – Square brackets are used to extract values from string constants.

In addition, we have non-overloaded operators:

(i:j) : used with arrays to denote substring of the array from index i to index j. If i is not specified, it is assumed to be from the beginning of the string; if j is not specified it is assumed to be till the end of string. If neither is specified an exception is thrown.

4. Project Plan

4.1 Team Responsibilities

Abhinav Saini	○ Programming Language Design ○ Code Generation ○ Writing part of the compiler file
Archana Balakrishnan	○ Programming Language Design ○ Writing a part of the parser ○ Coding some program features across program units ○ Testing, Debugging and Coding fixes across all program units ○ Writing a part of the document
Pradeep Dasigi	○ Programming Language Design ○ Writing the scanner ○ Completing the parser ○ Writing the compiler, and generating Java code
Raghavan Muthuregunathan	○ Language Idea ○ Programming Language Design ○ Testing (Functional and Non-functional)
Srilekha VK	○ Programming Language Design ○ Debugging the scanner ○ Writing the Final report

4.3 Development Environment

Dynamo was developed using Ocaml in Linux environment (Kernel version 2.6.32)

1. Ocamllex: An Ocaml tool for the scanner.
2. Ocamlyacc: An Ocaml tool for the parser.
3. Python : To run various testing scripts in Linux environment.

4.4 Development Tools

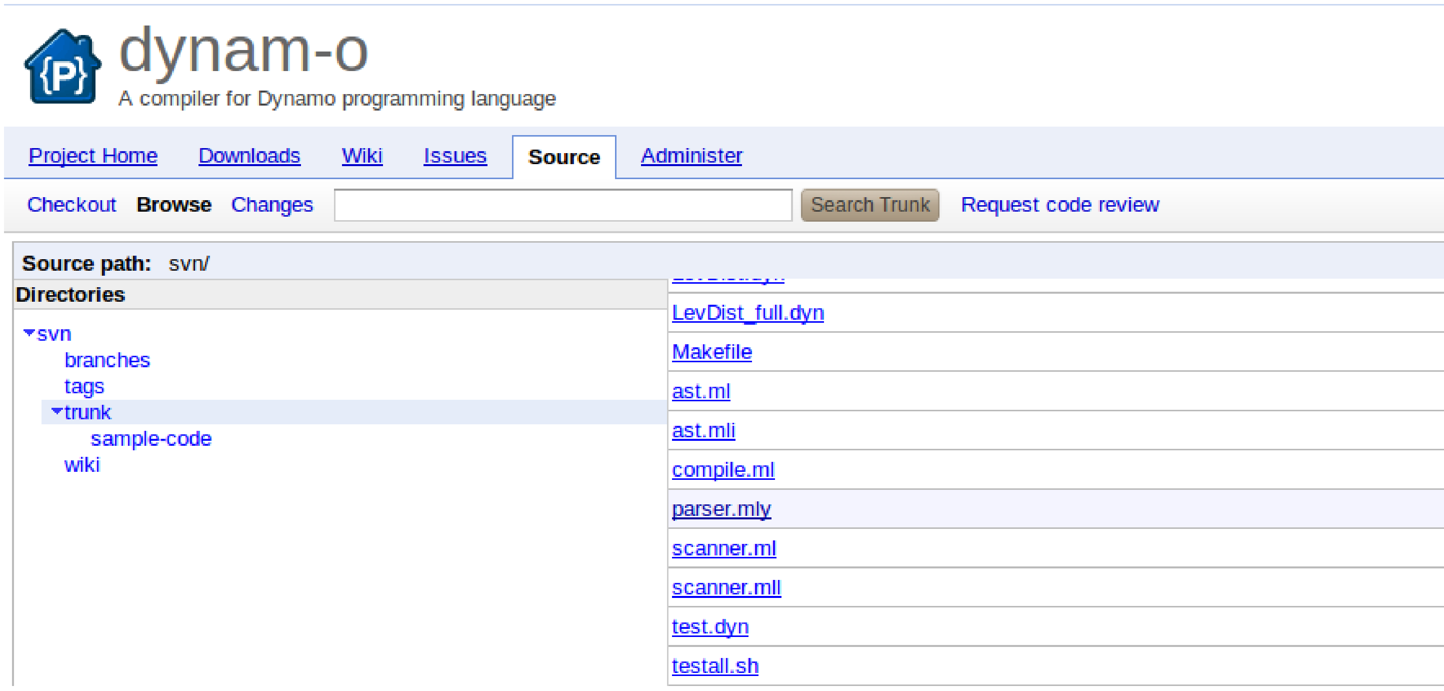
4.4.1 Google code repository:

Google code is Google’s official developer site. It features developer tools, technical resources and a project hosting service. This service provides revision control, offering both Subversion and Mercurial, an issue tracker, a wiki for documentation, and a file download feature. The service is available and free for all Open Source projects that are licensed under one of nine licenses (Apache, Artistic, BSD, GPLv2, GPLv3, LGPL, MIT, MPL and EPL).
Dynamo is licensed under the GNU GPL v3 license.

4.4.2 Version Control

The version control used in this project is Subversion. One can view the Subversion filesystem as "two-dimensional". Two coordinates are used to unambiguously address filesystem item: path and revision. Each revision in a Subversion filesystem has its own root, which is used to access contents at that revision. Files are stored as links to the most recent change; thus a Subversion repository is quite compact. The system consumes storage space proportional to the number of changes made, not to the number of revisions. The Subversion filesystem uses transactions to keep changes atomic. A transaction operates on a specified revision of the filesystem, not necessarily the latest. The transaction has its own *root*, on which changes are made. It is then either committed and becomes the latest revision, or is aborted. The transaction is actually a long-lived filesystem object; a client does not need to commit or abort a transaction itself, rather it can also begin a transaction, exit, and then can re-open the transaction and continue using it.

4.4.3 Directory Structure of Project



Source path: svn/

Directories

- svn
  - branches
  - tags
  - trunk
    - sample-code
  - wiki

LevDist_full.dyn
Makefile
ast.ml
ast.mli
compile.ml
parser.mly
scanner.ml
scanner.mll
test.dyn
testall.sh

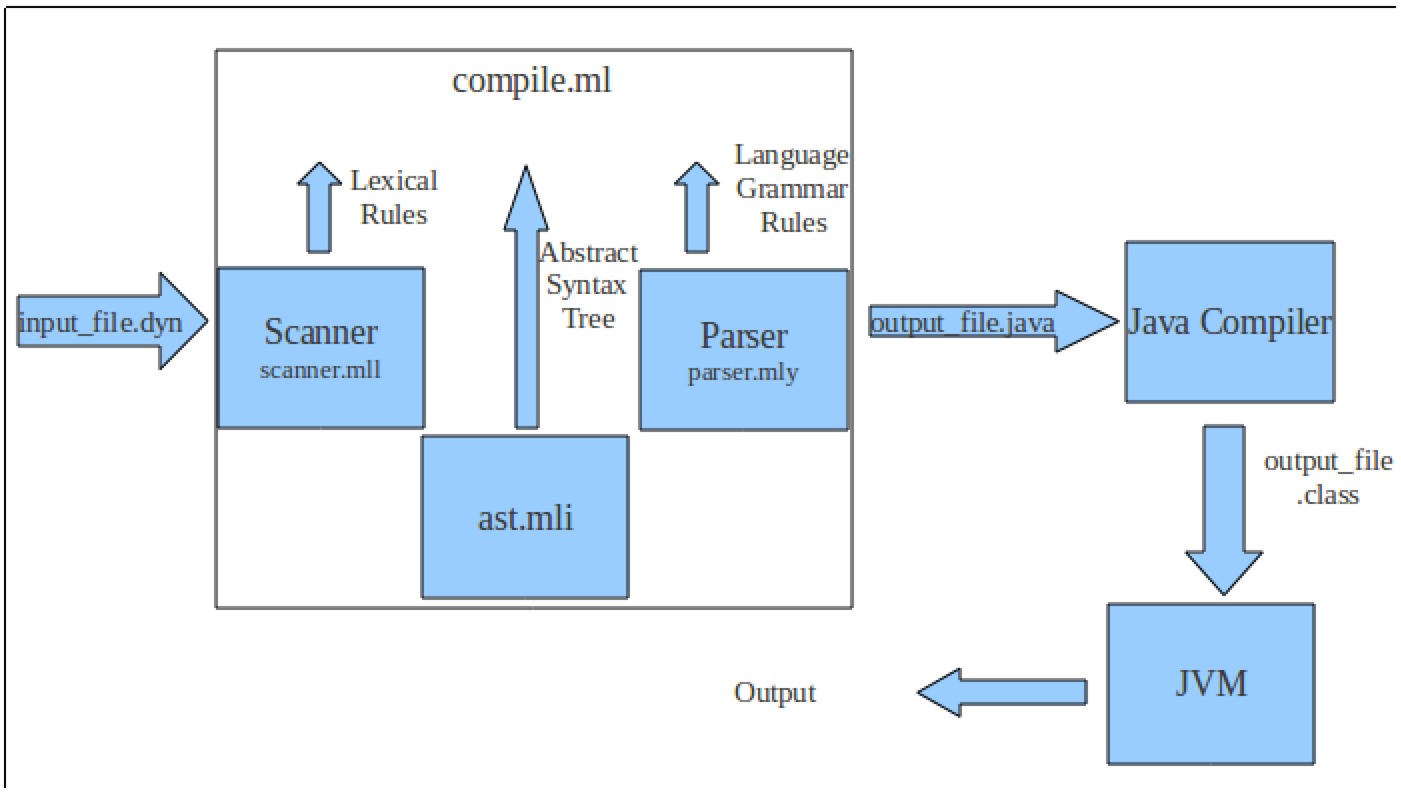
4.4.4 Project Timeline

Date	Milestone
22 September, 2010	Selecting Project Idea (Dynamic Programming)
27 September, 2010	Language Features
29 September, 2010	Submission of Project Proposal
22 October, 2010	Language Design
28 October, 2010	Discussion of Language Grammar
3 November, 2010	LRM Submission
1 December, 2010	Work on Scanner
1 December, 2010	Work on Parser
5 December, 2010	Scanner tested

8 December 2010	Parser Tested
21, November 2010	AST rules finalized
20, December 2010	AST created
22 December, 2010	Successfully mapped sample .dyn files to java
21, December 2010	Final Test cases generation
21, December 2010	Recurrence tests done
22, December 2010	Final Project Demo and Report Submission

5. Architectural Design

5.1 Components of the System



Lexical Scanner - Reads the tokens from the input .dyn program and passes them as tokens to

the parser
Parser - The parser with the help of the AST constructs a parse tree.
Compiler - The compiler takes as input the outputs of the lexical analyzer and the parser and converts the .dyn to the .Java code.
Abstract Syntax Tree - The AST contains the type definitions

The translated Java code requires Java compiler and JVM to run.

5.2 Grammar of the Language

The grammar of the language is specified in the Parser as shown in the appendix. But broadly, a Dynamo program maps to two main sections - Data Initialization and statements that form the logic of the code.
Data Initialization has a fixed syntax:
data-init <var_name>;

The logic of the program mainly consists of if-else constructs, function calls, and array accesses.

5.3 Code Generation
The mapping from *.dyn* files to *.java* files was dictated by the following rules:

- The number of data-initialization statements determine the dimensions of the memoization table. The lengths of the input strings correspond to the number of rows and columns. Thus these serve as the indices of the **for** loops in the translated .java file.
- Each *if* statements in the *.dyn* corresponds to filling entries in the table. For instance in the .dyn file, the statement 'if (str1=0 && str2=0) 0' which means that if both the input strings are empty,then return 0, would correspond to filling the table at the cell [0,0]
- Min and Max functions from the *.dyn* files generate the static methods min and max from Math class in Java. The recursion in the *.dyn* files is expressed in the forms of **for** loops.

6. Test Plan

The Testing was done as described below. The test scripts are attached in the Appendix.

Initial elementary tests were conducted with each module with negative cases and border cases as input to ensure that exceptions were thrown in desired situations.

Testing is divided into two types:

Non-functional testing:

1. Testing the dimension

As mentioned state of the DP problem is equal to number of data-init statements in the .dyn program. So a test case is designed to check if the dimension of memoization table is equal to number of data-init statements.

2. Testing for anti causality

DP solutions are computed bottom up. Hence DP solution cannot be anti causal ie. one cannot compute a solution for a sub problem *i* from an non existing solution to sub problem *i+1* . This anti causal property is implemented at compile time by analyzing the recurrence equation. This feature is tested.

3. Checking the constructs of the language

A test case is designed if the data-init statements, min , max are properly translated in the resultant java code

4. Checking the functionality of if-else

DP solution consists of If-Else statements to mention the base case and recursive case equation. A Test case is designed if its properly translated in the target java code.

5. Checking $-$ java statements

A support is present in the *.dyn* language to add embedded java code. This feature is tested if the $-$ statement is properly translated into target java code.

6 . Test case for analyzing the recurrence.

A Test case if designed to check if the various recurrence equations are properly translated in the target java code.

Functional Regression:

These are test cases in which functionality of the code is checked based on the output of reference java program and output produced by generated java program.

7. Lessons Learned

Archana:

She gained a much better understanding of the intricacies of programming languages - meaning behind rules, need for type-checking, the significance of methods like short-circuit evaluation, design features for writability, etc. These days, when she codes programs, she thinks from a compiler point-of-view.

Abhinav:

Had a good introduction to functional programming and the different components of a compiler.

Different aspects of Dynamic Programming like bottom up DP and top down DP.

Pradeep:
- Could gain an in-depth understanding of how compilers work and why a systematic procedure in developing them is important.
- Discovered the joy of using OCaml and functional programming languages in general

Raghavan
- I understood Dynamic Programming better than ever. Type Inference is one of the toughest components of compiler development.
- Recurrence Analysis to test Anti-Causal behaviour gave more insight about the nature of various recurrences.

Srilekha
- I learnt to appreciate Dynamic Programming and the need for simpler iterative code from the compiler's point of view.
- Now fully appreciates the simplicity, elegance and intuitiveness of functional programming and thinking
- Will stop taking aspects of a compiler like type inference etc. for granted. Realises the importance of language design (in addition to other aspects) in usability of a language.

8. Potential Enhancements

- Adding functionality for backtracking in which memoization is not inherent in the nature of the problems.
- Extending the solution space by allowing to solve for problems having more than 2 dimensions in the memoization table.

Appendix

Scanner

```
{ open Parser }

rule token = parse
 [' ' '\t' '\r' '\n'] { token lexbuf }
| "/*"        { comment lexbuf }
(*| '%' _ '%' as lxm       { JAVACODE(lxm.[1]) }*)
| ','         { COMMA }
| '('         { LPAREN }
| ')'         { RPAREN }
| '{'         { LBRACE }
| '}'         { RBRACE }
| '['         { LSQRBRACE }
| ']'         { RSQRBRACE }
| ';'         { SEMI }
| '+'         { PLUS }
| '-'         { MINUS }
| '*'         { TIMES }
| '/'         { DIVIDE }
| "="  { EQ }
| "!="        { NEQ }
| '<'         { LT }
| "<="        { LEQ }
| ">"         { GT }
| ">="        { GEQ }
| "&"          { AND }
| "|"          { OR  }
| '#'          { STRLEN }
| ':'          { ARROP }
| "if"        { IF }
| "else"   { ELSE }
| '$' [^'$']* '$' as lxm { JAVACODE(lxm) }
| "data-init"    { DATAINIT }
(*| "memoize"    { MEMOIZE }*)
| ['0'-'9']+ as lxm { LITERAL(int_of_string lxm) }
| ['a'-'z' 'A'-'Z']['a'-'z' 'A'-'Z' '0'-'9' '_']* as lxm { ID(lxm) }
| '"' ['a'-'z' 'A'-'Z']['a'-'z' 'A'-'Z' '0'-'9' '_']* '"' as lxm { WORD(lxm) }
| '"' ' ' '"' as lxm { WORD(lxm) }
| '"' '"' as lxm { WORD(lxm) }
| eof { EOF }
| _ as char { raise (Failure("illegal character " ^ Char.escaped char)) }

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

Parser

```
%{ open Ast %}

%token PLUS MINUS TIMES DIVIDE
%token EQ NEQ LT LEQ GT GEQ AND OR
%token LPAREN RPAREN LBRACE RBRACE LSQRBRACE RSQRBRACE SEMI COMMA
%token STRLEN
```

```
%token DATAINIT
%token IF ELSE
%token <int> LITERAL
%token <string> ID
%token <string> WORD
%token <string> JAVACODE
%token ARROP
%token EOF

%left ARROP
%left OR, AND
%left EQ, NEQ, LT, LEQ, GT, GEQ
%left PLUS, MINUS
%left TIMES, DIVIDE
%left STRLEN

%start programStruct
%type <Ast.programStruct> programStruct

%%

programStruct:
  ID LBRACE program RBRACE { ProgramBlock($1, $3) }

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

dataIntl_list:
      /* nothing */              { [] }
| dataIntl_list dataIntl     { $2 :: $1 }

dataIntl:
      DATAINIT ID SEMI { Dinit($2) }
| LBRACE dataIntl_list RBRACE   { DBlock(List.rev $2) }

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

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

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

expr:
      LITERAL       { Literal($1) }
| STRLEN expr       { Strlen($2) }
| ID                { Id($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) }
| 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) }
| expr AND    expr { Binop($1, And, $3) }
| expr OR     expr { Binop($1, Or, $3) }
| ID LSQRBRACE expr RSQRBRACE { StrInd($1, $3) }
| ID LSQRBRACE expr ARROP expr RSQRBRACE { StrRange($1, $3, $5) }
| ID LPAREN actuals_opt RPAREN { Call($1, $3) }
| LPAREN expr RPAREN { $2 }
| WORD       { Word($1) }

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

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

Ast.mli

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

type expr =
       Literal of int
| Id of string
| Binop of expr * op * expr
| Strlen of expr
| StrInd of string * expr
| StrRange of string * expr * expr
| Word of string
| Call of string * expr list
| Noexpr

type dataIntl =
       DBlock of dataIntl list
| Dinit of string

type stmt =
       Block of stmt list
| Expr of expr
| If of expr * stmt * stmt
| Copy of string
| Nostmt

type program = dataIntl list * stmt list

type programStruct = ProgramBlock of string * program
```

Compile.ml

```
open Ast

let rec string_of_expr = function
       Literal(l) -> string_of_int l
 | Id(s) -> s
 | Word(s) -> s
 | Strlen(s) -> string_of_expr s ^ ".length()"
 | Binop(e1, o, e2) ->
       string_of_expr e1  ^ " " ^
       (match o with
       | And -> "&&" | Or -> "||"
       |  Add -> "+" | Sub -> "-" | Mult -> "*" | Div -> "/"
       | Equal -> "==" | Neq -> "!="
       | Less -> "<" | Leq -> "<=" | Greater -> ">" | Geq -> ">=") ^ " " ^
       string_of_expr e2 ^ " "
 | StrInd(s, i) -> s ^ ".charAt(" ^ string_of_expr i ^ ")"

| StrRange(s, i1, i2) -> s ^ ".substring(" ^ string_of_expr i1 ^ ", " ^ string_of_expr
i2 ^ ")"
 | Call(f, el) ->
       (match f with
       | "min" -> "temp[i1][i2] = " ^ "Math.min" ^ "(" ^ String.concat ", " (List.map
string_of_expr el) ^ ")"
       | "max" -> "temp[i1][i2] = " ^ "Math.max" ^ "(" ^ String.concat ", " (List.map
string_of_expr el) ^ ")"
       | _ -> "temp" ^ "[" ^ string_of_expr (List.hd el) ^ "]" ^ "[" ^
(string_of_expr (List.hd(List.tl el))) ^ "]")
 | Noexpr -> ""

let dim = ref 3

let rec string_of_stmt = function
       Block(stmts) ->
       String.concat "" (List.map string_of_stmt stmts) ^
       let rec func i l = if(!i != 0) then (decr i; func i ("}"::l)) else ""::l in
String.concat "\n" (func dim [])
 | Expr(expr) -> string_of_expr expr ^ ";\n"
  | If(e, s1, s2) -> "if (" ^ ( if (string_of_expr e) = "str1 == \"\" " then
("i1 == 0") else (if (string_of_expr e) = "str2 == \"\" " then "i2 == 0" else (if
(string_of_expr e) = "str1 == \"\" && str2 == \"\" " then "i1 == 0 && i2 == 0" else
(if (string_of_expr e) = "str1 == \"\" || str2 == \"\" " then "i1 == 0 || i2 == 0"
else (string_of_expr e))))) ^ ")\n{\n" ^
       "temp[i1][i2] = " ^ string_of_stmt s1 ^ "}\n" ^ "else\n{\n" ^ string_of_stmt s2
^ "}\n"
 | Copy(s) -> String.sub s 1 ((String.length s) - 2) ^ "\n"
 | Nostmt -> ""


let rec string_of_dataIntl = function
       DBlock(inits) ->   let func i = "String "^
                   i in String.concat "," (List.map func (List.map string_of_dataIntl
inits)) ^ ")\n{\n" ^
                   let dimCount = ref 0 in let func i = "int temp" ^ string_of_int
(ignore(incr dimCount); !dimCount)
                   ^ "=" ^ " " ^ string_of_dataIntl i ^ ".length()+1;\n" in
String.concat "" (List.map func inits) ^ "int temp" ^
                   let func i = "[]" in String.concat "" (List.map func inits) ^ " =
new int" ^
                   let dimCount = ref 0 in let func i = "[temp" ^ string_of_int
(ignore(incr dimCount); !dimCount)
                   ^ "]" in String.concat "" (List.map func inits) ^ ";\n" ^
                   let dimCount1 = ref 0 and dimCount2 = ref 0 and dimCount3 = ref 0
and dimCount4 = ref 0 in
                   let func i = "for(int i" ^ string_of_int (ignore(incr dimCount1); !
dimCount1) ^ " = 0; i"
                   ^ string_of_int (ignore(incr dimCount2); !dimCount2) ^ " < temp" ^
                   string_of_int (ignore(incr dimCount3); !dimCount3) ^ " ; i" ^
string_of_int (ignore(incr dimCount4); !dimCount4)
                   ^ "++)\n{\n" in String.concat "" (List.map func inits)
 | Dinit(init) ->  init

let string_of_program(intls, logsts) =
 String.concat "" (List.map string_of_dataIntl intls) ^ "\n" ^
 String.concat "" (List.map string_of_stmt logsts)

let string_of_programStruct = function
```

```
      ProgramBlock(id,  code)  ->  "import  java.lang.Math;\npublic  class  "  ^  id
^ "Class\n" ^ "{" ^ "\npublic static void " ^ id
                                 ^  "("  ^  string_of_program  code  ^  "\npublic  static
void main(String args[])\n"
                                 ^"{\n" ^ "System.out.println(" ^ id ^ "(" ^ "args[0],
args[1]"  ^ "));" ^ "\n}" ^ "\n}"

let _ =
 let in_channel = open_in Sys.argv.(1) and out_channel = open_out Sys.argv.(2)in
 let lexbuf = Lexing.from_channel in_channel in
 let programStruct = Parser.programStruct Scanner.token lexbuf in
 output_string out_channel (string_of_programStruct programStruct)
```

Testing Scripts

test_1.py

```
import sys;
keyword = dict();
f = open(sys.argv[1], "r");
for line in f:
      if "data-init" in line:
            words = line.split();
            words[1] = words[1][0:len(words[1])-1];
            keyword[words[1]] = 0;
f.close();
f = open(sys.argv[2], "r");
newkeywords = dict();
for i in keyword:
      newkeywords[i] = keyword[i];
for line in f:
      for S in keyword:
            words = line.split();
            for w in words:
                  if S in w:
                        newkeywords[S]=1;
kflag =0;
f.close();
for k in keyword:
      if (newkeywords[k] == 0):
            print "keyword ",k," is missing";
            kflag = 1;
            print "Test  FAILED";
if (kflag == 0):
      print "TEST PASSED";
```

test_2.py

```
import sys;
java = dict();
f = open(sys.argv[1], "r");
for line in f.readlines():

      line = line.lstrip();
      line = line.rstrip();
      if ('$' in line):
            line = line.lstrip();
            line = line.rstrip();
            line=line[1:len(line)-1];
            line = line.lstrip();
```

```
              line = line.rstrip();
              java[line] = 0;

f.close();
f = open(sys.argv[2], "r");

newkeywords = dict();

for i in java:
       newkeywords[i] = java[i];

for line in f:
       line = line.lstrip();
       line = line.rstrip();
       for S in java:
              if S in line:
                     newkeywords[S] = 1;
                     if line in S:
                            newkeywords[S] = 1;

kflag =0;
f.close();

for k in java:
       if (newkeywords[k] == 0):
              print "java statement ",k," is missing";
              kflag = 1;
              print "Test  FAILED";
if (kflag == 0):
              print "TEST PASSED";
```
test_3.py
```
import sys;
d = dict();
f = open(sys.argv[1], "r");
d["if"]=0;
d["else if"] =0;
d["else"] = 0;

for line in f:
       if "else if" in line:
              if (d.__contains__("else if")):
                     d["else if"] +=1;
              else:
                     d["else if"] = 1;
              elif "if" in line:
                     if (d.__contains__("if")):
                            d["if"] += 1;
                     else:
                            d["if"] = 1;
              elif "else" in line:
                     if (d.__contains__("else")):
                            d["else"] +=1;
                     else:
                            d["else"] = 1;
f.close();
f = open(sys.argv[2], "r");
s = dict();
s["if"]=0;
s["else if"] =0;
s["else"] = 0;
for line in f:
       if "else if" in line:
              if (s.__contains__("else if")):
                    s["else if"] +=1;
            else:
                    s["else if"] = 1;
        elif "if" in line:
            if (s.__contains__("if")):
                    s["if"] += 1;
            else:
                    s["if"] = 1;
        elif "else" in line:
            if (s.__contains__("else")):
                    s["else"] +=1;
            else:
                    s["else"] = 1;
kflag = 0;
for i in s:
            if (s[i] != d[i]):
                    kflag = 1;
            print "Test Failed for mismatch in count of ",i;
if kflag ==0:
            print "TEST PASSED";
```

Test_4.py

```
## Recurrence Check
import sys;
f =open(sys.argv[1], "r");
first_line_flag = 1;
KEY =""
for line in f:
                        if first_line_flag == 1:
                                                first_line_flag = 0;

                                                word = line;
                                                word = word.lstrip();
                                                word = word.rstrip();
                                                KEY = word;
f.close();
f = open(sys.argv[1], "r");
total_count = 0;

for line in f.readlines():
                        line = line.lstrip();
                        line = line.rstrip();

                        if KEY in line:
                                                total_count +=1;
f.close();

f = open(sys.argv[2], "r");
check_count = 0;

for line in f:
                        line  = line.lstrip();
                        line = line.rstrip();

                        if KEY in line:
                                                check_count +=1;
if (check_count == total_count):
                        print "TEST PASSED";
else:
                        print "TEST FAILED - mismatch in recurrences";
```

Anti Causal Check

```
import sys;
import re;
f = open(sys.argv[1], "r");
```

```
DP = "";
for line in f.readlines():
       line = line.lstrip();
       line = line.rstrip();
       DP = line;
       break;
f.close();
print DP;
##############
Keyword = set();
f = open(sys.argv[1], "r");
for line in f.readlines():
       line = line.lstrip();
       line = line.rstrip();
       if ("data-init" in line):
               words = line.split();
               word = words[1][0:len(words[1])-1];
               Keyword.add(word);
###################
##do  regular expression check
antiCausal = 0;
for item in Keyword:
       patternString = item+"(\[.+\])";
       f = open(sys.argv[1],"r");
       for line in f.readlines():
               m = re.search(patternString, line);
               if m:
                       g = m.group(1);
                       if "+" in g:
                               print "Warning: your DP Recurrence Looks Anti Causal";
                               print g, "in", line;
                               antiCausal = 1;
       f.close();
for item in Keyword:
       patternString = item+"(\(.+\))";
       f = open(sys.argv[1],"r");
       for line in f.readlines():
               m = re.search(patternString, line);
               if m:
                       g = m.group(1);
                       STR = g[g.index('('):g.index(')')];
                       if "+" in STR:
                               print "Warning: your DP Recurrence Looks Anti Causal";
                               print STR, "in", line;
                               antiCausal = 1;
       f.close();


if antiCausal == 0:
       print "Anti Causality Test is done. No doubts about Anti Causal behaviour"

import sys;
import re;
f = open(sys.argv[1], "r");
DP = "";
for line in f.readlines():
       line = line.lstrip();
       line = line.rstrip();
       DP = line;
       break;
f.close();
print DP;
##############
```

```
Keyword = set();
f = open(sys.argv[1], "r");
for line in f.readlines():
        line = line.lstrip();
        line = line.rstrip();
        if ("data-init" in line):
                words = line.split();
                word = words[1][0:len(words[1])-1];
                Keyword.add(word);
###################
##do  regular expression check
f.close();
antiCausal = 0;
patternString = DP+"(\(.*\))";
f = open(sys.argv[1], "r");
kflag = 1;
for line in f.readlines():
        m = re.search(patternString, line);
        if m:
                G = m.group(1);
                for k in G:
                        if k not in G:
                                kflag = 0;
                                print k, "term is missing in ", G;
if kflag:
        print "Recursion is perfect";
```

test all.sh

```
echo $1,$2
echo "Test Case 1";
python test_1.py $1 $2;
echo "Test case 2:";
python test_2.py $1 $2
echo "Test Case 3:";
python test_3.py $1 $2;
echo "Test Case 4:";
python test_4.py $1 $2;
echo "TEST case 5:"
python test_5.py $1;
echo "Test Case 6";
python test_6.py $1;
echo "Compiling ",$2;
javac $2;
```