भारतीय प्रौद्योगिकी संस्थान हैदराबाद
Indian Institute of Technology Hyderabad

Compilers 2

Tureasy

Language Report

By IITH8

INDEX

1. Abstract

In this document we propose Tureasy a general purpose language. Tureasy is designed to provide optimizations and help for discrete math. Tureasy supports paradigms like “declarative”, ”imperative”, “procedural” employed to particular sections of the concerned domain. To help computer scientists and mathematicians in coding their algorithms, Tureasy provides rich libraries of built-in functions and data structures supporting various problems in discrete math.

Tureasy is a general purpose programming language designed by team IITH8 in 2020 as a part of Compilers project. The language got its name from two words Turing and Easy. The word Turing is given in honour to Alan Turing, the founder of the famous imitation game (Turing Test). The word Easy is given to signify the ease of programming in this language.

2. Motivation:

The basic idea which motivated the language design in the beginning was even dumbest programmers should be able to improve writing code in our language. This made us come up with an idea called smart compiler which is more than a mere translator. There are many other languages that provide certain improvements in code internally, but the programmer seldom has a role to play in it. In Tureasy language, a concept called tags was introduced that was improved over time. These tags analyse the code and provide users with appropriate suggestions.Tureasy language that would collect data from the user (with his consent) and improve its models over years.

Discrete mathematics plays a central role in the fields of modern cryptography, social networking, digital signal and image processing, computational physics, analysis of algorithms, etc. as more and more mathematics that is done, both in academia and in industry, is discrete. This motivates us to build a programming language focused on discrete mathematics, to help computer scientists and mathematicians to work more easily and efficiently as compared to that while using a General Purpose Language (GPL).

3. Tutorial

3.1 Installation

Before building our compiler make sure that your computer meets the following requirements.

- Ocaml must be installed using https://dev.realworldocaml.org/install.html or via terminal -

```
$ sudo apt-get install ocaml -y
$ sudo opam init -y
$ eval `opam config env`
$ sudo apt install ocaml-nox -y
```

- LLVM must be installed using https://llvm.org/ or via terminal -

```
$ sudo apt install llvm llvm-runtime m4
$ opam install llvm
```

3.2 Building our compiler

To build our compiler, the following commands need to be executed in the root directory.

```
$ make
$ make clean        # to clean all the files built
```

3.3 Basic syntax

Ex. 3.3.1

```
def void main()
{
    string message= "Hello"; $ This declares and defines a string
literal
    int a=1;
    float b=0.1;
    $* Int-float type compatibility is supported *$
    a=a+b;
    print(message);
}
```

Every program must have a main function. Every function definition starts with the keyword “def”, followed by return type, then function name and argument list in parentheses. Types of arguments must be specified. Multi-line comments are enclosed in $* and *$, while single-line comments start with $. Int-float type compatibility is supported, so typecasting is not required. In the example above, the final value of a is still 1 as the decimal part is truncated when assigning to an integer.

3.4 Running your first program

After the compiler is built and you have your first program ready, you can compile and execute it using the following commands.

- $ ./tureasy.native <filename>.tz
- ./a.out

4. Language Reference Manual

4.1. Data Types

4.1.1 Primitive Data Types:

data type Name	**Description**	**Initialization**
int	64-bit signed integer value	int x = 7;
float	64-bit signed floating-point number	`int x = 7;`

string	sequence of characters	string str1 = "Testing 1..2..3... Hello world! ";
bool	stores either 0/1 values for true/false	bool a = 1; bool b = 0;
struct	user-defined, similar to a struct in C	struct ex1 { int x; float y; string str1; matrix int Q; set B; }; struct ex1 myStruct.str1 = "Example";

4.1.2 Non-Primitive Data Types:

Using [ ] for declaring elements of graph, set, matrix, etc.

data type Name	**Description**	**Initialization**
numset	stores a set of numerical values	`numset A = [1, 2, 3, 8.5, -1];`
strset	stores a set of strings	`strset A = [“1”, 2”,“3”, "str","popl"];`
graph	a set of nodes and edges	`graph G = [1 : 2,3,4; 2 : 1,5; 3 : 1,4; 4 : 1,3; 5 : 2,6; 6 : 5; ];`
matrix	a matrix	`matrix M = [[1.0,4,6], [0.5,3,0]];`

4.2. Lexical Conventions

4.2.1 Comments:

Both single and multiline comments are supported in Tureasy.
All tokens after a **$** symbol on a line are considered to be part of a comment and are ignored by the compiler.

Ex. 4.2.1

```
$ This is a single-line comment.
matrix  M1 = [[1.0, 4, 6],  $ This is also a valid single line
comment
              [0.5, 3, 0],
              [9.1, 2, 7]];
```

Multiline comments start with **$*** and end with ***$**.

Ex. 4.2.2

```
matrix  M1 = [[1.0, 4, 6]
              [0.5, 3, 0] $* This is a valid comment *$
              [9.1, 2, 7]];

matrix M2 = [[1.0, 4, 6],
             $* This is a valid
                multiline comment *$
             [0.5, 3, 0],
             [9.1, 2, 7]];
```

Nesting comments are not allowed, and comments cannot be inside strings.

Ex.4. 2.3

```
$* Comments cannot be $* nested *$ like this *$
```

```
string str1 = "Hello $* This is also part of str1 *$ world";
matrix  M1 = [[1.0, 4, 6]
              [0.5, 3, 0]
              [9.1, 2, 7]];
```

4.2.2 Whitespace:

Whitespace, including tabs, spaces, and comments, will be ignored, except in places where spaces are required for the separation of tokens. Tabs/indentation cannot be used to define the scope in Tureasy.
Both of the programs in Ex 2.4 and 2.5 below are equivalent:
Ex.4. 2.4

```
void main ()
{
    int a;
    a = input();
    if(a%2 == 0)
    {
        a=a+5;
        output(a);
    }
    else
    {
        output(a);
    }
    return;
}
```

Ex. 4.2.5

```
void main()
{int a;
a=input();
if(a%2 == 0)
{a=a+5;
output(a);}
else{output(a);}
return;}
```

4.2.3 Reserved Keywords:

The following words are reserved keywords in Tureasy and they cannot be used as regular identifiers.

if	else	switch	case	default	link
loop	int	return	break	continue	struct
	rename	bool	void	const	null
int		float	string	numset	strset
graph	matrix	AND	OR	const	

Apart from the ones listed above, all the data types listed in section 3 are also reserved keywords.

4.2.4 Identifiers:

Identifiers must start with a letter, which can be followed by a sequence of letters, digits, and underscores. Hyphens and other special characters cannot be present in an identifier. Tureasy is case-sensitive, so the identifiers foo and FOO and Foo and fOo are distinct.

Ex.4. 2.6

```
foo;        $ correct
fo0;        $ correct
f_o;        $ correct
_foo;       $ correct;
```

Ex.4. 2.7

```
0of;        $ not a proper identifier
f-o0;       $ not a proper identifier
fo^;        $ not a proper identifier
```

4.2.5 Punctuators:

Statements should be terminated with semicolons (;).

4.2.6 Tags:

Tags in Tureasy are used to identify the part of code that requires specific modifications. The nodes between the opening and closing tags are colored in the abstract syntax tree formed after the semantical analysis. They begin with **#** and end with **#!**. They should not be used between instructions, string literals, integers, etc.

4.3. Program structure

4.3.1 Declarations

A program consists of various entities such as variables, functions, types. Each of these entities must be declared before they can be used.

Ex.

```
int fun1(int num1)
{
    return num1 + 42;
}

void main()
{
    int num1;
    num1 = fun1(2);
    string str1 = "Number is ";
    output(str1, num1);
}
```

OUTPUT:

```
        Number is 44
```

4.3.1.1 Variable declaration:

All variables used in a program must be declared before using them in a statement, followed by a semicolon.

- int w = 5, x;
- bool y;
- float z;
- set A;
- string s;
- matrix M1;
- graph g;

4.3.1.2 Declaration scope:

In Tureasy, the scope is defined by using { }. The identifier introduced by a declaration is valid only within the scope where the declaration occurs. Variables declared in global scope must have unique identifiers. The same identifier cannot be used to refer to more than one entity in a given local scope. For example, in Ex. 5.1 above, the variable num1 refers to two different variables, one in main and the other in the scope of fun1(). However, the programs below in Ex 5.2 and 5.3 are not valid.

Ex.

```
int fun1(int x)
{
}

int num1;

int main()
{
    int num1; $ invalid identifier as num1 has already been
used for a global variable
    num1 = fun1(2);
    string str1 = "Number is";
    output(str1, num1);
}

int fun1(int num1)
{   $ invalid identifier as num1 has already been used for a
global variable
    return num1 + 42;
}
```

Ex.

```
int fun1(int num1) $ invalid identifier as it has already been used
for a global variable
{
    return num1 + 42;
}

int num1;

void main()
{
    int num2;
    float num2; $ invalid identifier as num2 has already been used
for a variable within the same/higher scope
    num1 = fun1(2);
    string str1 = "Number is";
    output(str1, num1);}
```

4.3.1.3 Rename:

Rename keyword is used to declare a new name that is an alias for another name, the new name is taken to be everything after the comma on the same line. It is similar to the keyword typdef in C/C++.

Ex.

```
struct node
{ int x;
  float y;
  string str1;
  matrix Q;
  set B;
};
rename struct node, node;  $ from this point onwards, we can use
'node' instead of 'struct node'
```

```
node ex1;
```

```
ex1.str1 = "Example";
```

4.3.1.4 const:

The const keyword specifies that a variable's value is constant and tells the compiler to prevent the programmer from modifying it. For instance, the program below in Ex. 3.13 will not compile as the variable i should not be modified.

Ex.

```
int main()
{
   const int i = 5;
   i++;
}
```

4.3.1.5 Array declaration:

We use a one-dimensional matrix instead of a traditional array. If the elements are not initialized, then it is inferred to be a one-dimensional array. The size of the matrix can be specified by built-in functions. Default values of the matrix are of float type.

The syntax for matrix is

```
matrix variableName = [[...], [...], ...];  $ Here, ... indicates
comma separated values
```

4.3.1.6 Function declaration:

The syntax for function declaration is

```
return_type functionName(datatype_1 arg_1, datatype_2 arg_2, ...);
$ datatype_i is the datatype of arg_i
```

A function may or may not have arguments but the return type is a must (return type void can be used if none is required).

4.3.2 Initializers:

When an object is declared, its init-declarator may specify an initial value for the identifier being declared. The initializer is preceded by = and is either an expression or a list of initializers nested in braces.

4.3.2.1 Types of Initializations:

Default initialization:

When variables are declared but not initialized, they are initialized by default to zero in the case of primitive data types, an empty string for strings, and null for non-primitive data types. Non-primitive data types should be used without initialization.

Ex. 5.6

```
int a; $ Here, a = 0
set A; $ Here, A is null
matrix mtx;  $ Here, mtx is null
```

Direct initialization:

Ex. 5.7

```
int i = 3;
string s = "hello";
matrix mtx = [[2,3], [4,5]];
set A= [1,2];                    $ set
```

For matrices, the initializer is a square-bracket enclosed list of initializers for its members. If the array has an unknown size, the number of initializers determines the size of the array, and its type becomes complete. If the array has a fixed size, the number of initializers may not exceed the number of members of the array; if there are fewer, the trailing members are initialized to 0 or null, the default value of the type of the matrix.

Copy initialization:

It is the initialization of one variable using another variable. This kind of initialization can be used for all supported data types.

Ex. 5.8

```
int a = 3, b;
b = a;
```

4.3.3 Control Flow:

4.3.3.1 Conditional Statements:

If statements consist of a condition (an expression) and a series of statements. The series of statements is evaluated if the condition evaluates to True. If the condition evaluates to False, either the program continues or an optional else clause is executed. Below is the pseudocode :

```
if (condition(statement)) {
Statements;
}

else if (condition(statement)){
Statements;
}

else {
Statements;
}
```

Statement following if/ else if (condition) must be a boolean statement, or can be evaluated to boolean.

Ex.

```
numset A = {1,2,3,4,5};
```

```
if(size(A) == 5){
     output("the size is ", size(A));
     output(A);
}
```

4.3.3.2 Loop Statements:

Loop statement consists of a condition and incrementer separated by; followed by a series of statements. The statements are repeatedly evaluated as long as the condition remains True before each repetition. The “;” is used as a separator inside the condition statement and especially there is no restriction on using it as you can infer from the two examples below.
Note that assigning over the looping variable will not change the original variable in the scope.

Ex.

```
int n=0;
loop(isprime(n) AND n < 10000)
{
     n++;
     output("n is", n);
}
```

Ex.

```
int n = 0;
loop (n!=20;n++)
{
}
```

4.3.3.2 break statement:

- The break statement ends the loop immediately when it is encountered. Its syntax is: break;
- The break statement is almost always used with an if...else statement inside the loop.

Ex. :

```
 loop(i-- >= 0)
{
    x=func(i);
    if(x==2)
      x=x+2;
    else
      break;
}
```

4.3.3.3 continue statement:

- The *continue* statement skips the current iteration of the loop and continues with the next iteration. Its syntax is: *continue;*
- The *continue* statement is almost always used with the if...else statement.

Ex. :

```
 loop(i-- >= 0)
{
    x=func(i);
    if(x==2)
        continue;
    x=x+2;
}
```

4.4 Expressions

4.4.1 Operators Precedence - Highest to Lowest:

Purpose	**Symbol**	**Associativity**	**Valid Operands**
Parentheses for grouping of operations	( )	left to right	int, float
Member access operator	.	Left to right	struct

Unary negation	!	right to left	all bool
Shift operators	<<	left to right	Int, long
	>>	left to right	Int, long
Exponent	^	left to right	int, float, matrix
Modulo	%	left to right	int, float
Multiplication	*	left to right	int, float, matrix
Division	/	left to right	int, float
Addition	+	left to right	int, float, string, matrix
Subtraction	-	left to right	int, float
Union	|	left to right	numset, strset
Intersection	&	left to right	numset, strset
Set difference	~	left to right	numset, strset
Relational Operators			
	<= < >= >	left to right	all datatypes
	== !=	left to right	all datatypes
Logical Operators	AND OR	left to right	bool
Assignment operators	= *= += /= ^= %=	right to left	wherever the operator is valid

Ex.

```
        int a = 5+13;                 $ evaluates to 18
        float b = 15.23+5;            $ evaluates to 20.23
        float c = 4-10.2;             $ evaluates to -6.2
        int d = 5*6;                  $ evaluates to 30
        float e = 20.0 * 0.25;          $ evaluates to 5.0
        int f = 10/2;                   $ evaluates to 5
        float g = 10.8/2;               $ evaluates to 5.4
        int h = 13%3;                   $ evaluates to 1
        float i = 10.8%2;               $ evaluates to 0.8
        int j = 2^3;                    $ evaluates to 8
        float k = 3.5^2;                $ evaluates to 12.25
```

Ex.

```
 string str1 = "Hello";
 string str2 = "World";
 string str3 = str1 + str2;   $ "Hello"+"World" results "HelloWorld"
 string str4 = str2 + str1;   $ gives "WorldHello"
```

Ex.

```
        matrix M1 = [[2, 3, 4],
                     [1, 5, 0],
                     [1, 4, 8]];

        matrix M2 = [[1.0, 4, 6],
                     [0.5, 3, 0],
                     [9.1, 2, 7]];

        matrix Mat1 = [[5, 2, 3],
                       [8, 11, 0]];

        matrix M3 = M1 + M2;      $ matrix addition, term by term
        matrix M4 = M1 - M2;      $ matrix subtraction, term by term
        matrix M5 = M1 - M2;     $ matrix subtraction, decimal
truncated
        matrix M6 = M1 * M2;     $ matrix multiplication
        matrix M7 = M1 * M2;     $ matrix multiplication, truncates
fractional part
        matrix M1 ^= 2;            $ assigns M1^2 to M1
```

The assignment operator can be used on variables with the same data type. Int and Float data types are compatible with each other with respect to assignment, that is an int can be assigned to a float without getting a compilation error, and a float can be assigned to an int which leads to the decimal part being truncated in the int variable. Apart from int and float, variables of different data types cannot be assigned to each other.

4.4.2 Tag Expressions:

They mark the beginning and end of tags. They use the operator **#** for beginning and end with **#!**. The data between these expressions undergo analysis during compilation.

4.4.3 Expression Statements:

Statements are executed for their effect and do not have values. Each expression statement includes one expression, which is usually an assignment or a function call. In Tureasy, every expression is followed by a semicolon ";" .

Syntax:

```
expression;
```

In Tureasy, they fall into several categories like:

- Compound statements
- Labeled-statements
- Selection statements
- Conditional statements
- Iteration statements
- Jump statements

4.4.4 Compound Statements:

A compound statement is a sequence of statements enclosed by braces as follow:

Syntax:

```
{
statement1;
statement2;
}
```

If a variable is declared in a compound statement, then the scope of this variable is limited to this statement.

4.5. Tags

The tags are special statements that group parts of code that have some standard implementation involved. It is used in this format **#<tag_name> code #!<tag_name>**

Some of the commonly used public tags in Tureasy are

loop	Unique var	var < (const)
graph-theory	number-theory	var in range (range)
unused	dp	innerloop
checkhere	exponentiation	divide and conquer

The tags provide programmers with tips related to

1. Parallelism: There are some constructs of Tureasy which support parallel execution. The programs including such constructs would improve performance but are hard to code. The tags analyze the data and provide suitable constructs that could replace the existing code.
2. Constraints: The correctness of algorithms can be determined by finding base rules which must be satisfied throughout it. In large programs, it becomes practically impossible to keep track of these rules. So, the programmer could make use of constraint tags and the tag ensures that the property is maintained. In case of failure, it would suggest modifications for the same.
3. Memory optimization: The tags provide us with tips that could optimize memory too. There could be instances where the programmer might allocate heap memory but never use it or might use a lot of stack memory unnecessarily.

4. Time complexity: Tureasy tags try to improve the code by understanding the code and providing us with better constructs that could help us reduce time complexity.

The tags use turzers during compilation to provide these tips. The tags are used to color the nodes in the abstract syntax tree during the semantic analysis phase. During the compilation, an abstract syntax tree is formed after the semantic analysis phase which undergoes machine-independent code improvement. The turzers are used during this phase.

Turzers are the files that contain the machine learning models for analyzing the data between tags. The abstract syntax tree is used as input for the turzers. The models within turzers are made using trees and its traversal has a certain cost associated with it. The program is compared with these models and tips are given accordingly.

There are private turzers and private tags associated with companies. These turzers have an additional requirement of .tcnf files which are the turzer configuration files. These tags can be uniquely defined and modified by the company based on its requirements.

4.6. Built-ins and Standard Library Functions

We plan to implement some standard libraries/headers to provide support for the following domains:

1. Matrix operations
2. Graph operations
3. Common math functions
4. Set theory
5. String operations

5 Project plan

5.1 Process

We had weekly hour-long meetings every Thursday right after class to identify what additional features we could implement based on the contents taught during

classes of the week. During finals week, we met more often and worked more effectively trying to implement the features we concluded on .
Since we had to update our status report every week , we were not out of motivation any time in the whole process of building a compiler for our language. The deadlines of every phase of compiler involved submissions of presentation video of overview of that stage, demo video showing its working and the source code that was written for that stage subjected to our language syntax. This helped us understand the drawbacks and the improvements that we could bring up to our compiler overall.

5.2 Responsibilities

Note that our responsibilities sort of blurred in the middle where everyone became involved in everything . Here is the breakdown .

Nisha M	Nisha0212	System Tester
Gayala Manoj	manojgayala	Language Guru
Sharanya Gupta	shrx11	System Architect
Gubbala Suraj	Prime-Nemesis	System Architect
Sravanthi Reddy M	sravanthiM1	System Integrator
Vedika Verma	Vedika28-code	Project Manager
Jatin Kumar	jatinKumar-k	System Tester

5.3 Project Timeline

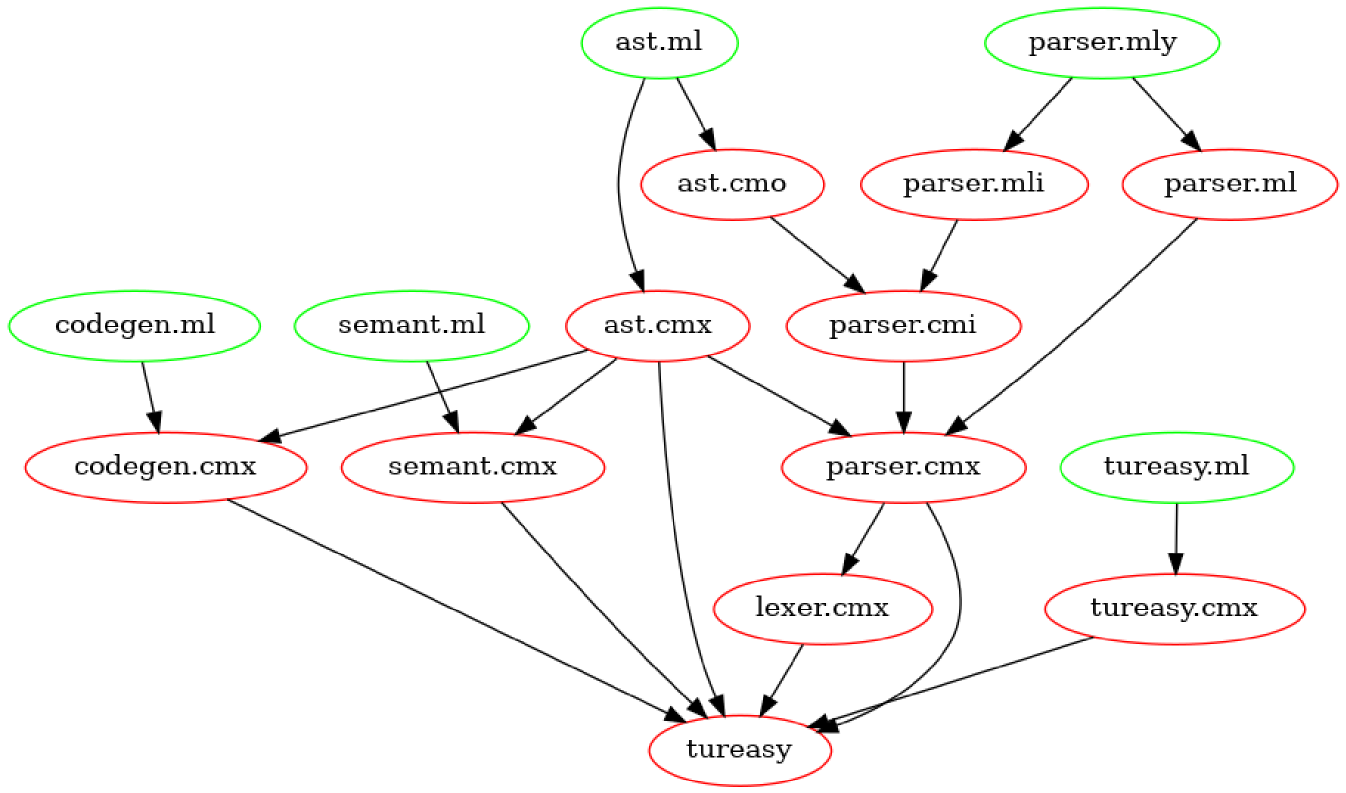
5.4 Development Environment

●Environment:
- ○ VS Code — development and collaboration tools
- ○ Github — main, development (“hello”), and testing branches.
- ○ Linux- to run and test the codes.

- Languages
  - Ocaml — primary development language
  - Bash — shell scripting to ease testing and development
  - LLVM 10.0 — target language
- Testing
  - Menhir + other Ocaml debugging tools — parser testing
  - Bash + personal testing files — for incremental development

6 Architecture Design

6.1 Architecture Diagram



6.2 Program Structure

1. Tureasy.ml

This is the upper level compiler that executes scanner, parser, semant, and codegen in order to generate the LLVM IR

2. parser.mly

This is the parser definition used by ocamlyacc to generate the parser that parses the scanned tokens
to produce an ast

3. ast.ml

This is the abstract syntax tree

4. lexer.mll

This is the scanner that turns the program into tokens.

5. semant.ml

This is the semantic checker that takes in the ast and returns a sast to ensure proper typing.

6. codegen.ml

The file takes in an AST and translates the ast into LLVM IR using the LLVM ocaml bindings.

7 Appendix

7.1 References

7.2 Code

1. Tureasy.ml

```
(* Lexer file in OCamllex for Tureasy *)

{
   open Lexing
   open Parser
   open Printf       (* error reporting *)

   let linecounter = ref 1

   exception SyntaxError of string

   let error character message lnum =
   sprintf "line %d: %s : %s" lnum character message

   let syntax_error lexbuf message lnum =
   raise ( SyntaxError(error (lexeme lexbuf) (message) (lnum)) )

}

let letter  = ['A'-'Z' 'a'-'z']
let digit = ['0'-'9']

rule token = parse
 [' ' '\t' '\r']                     { token lexbuf }        (* whitespace *)
| ['\n']                              { incr linecounter ; token lexbuf }
| "$*"                                { ml_comment lexbuf }    (* multi-line
comments *)
| "$"                                 { sl_comment lexbuf }    (* single-line
comments *)
(*                    Syntax                    *)
| '{'                                 { LBRACE }
| '}'                                 { RBRACE }
| '('                                 { LPAREN }
| ')'                                 { RPAREN }
| '['                                 { LBRACK }
| ']'                                 { RBRACK }
| ';'                                  { SEMICOLON }
| '.'                                  { DOT }
| ':'                                  { COLON }
| ','                                  { COMMA }
| "#"  (letter(letter|digit)* as tag)  { TAG_BEGIN(tag) }
| "#!" (letter(letter|digit)* as tag)  { TAG_END(tag)    }
(*                 Operators                 *)
| '!'                                  { NOT       }
| "++"                                 { INCR       }
| "--"                                 { DECR       }
| "<<"                                 { LSHIFT     }
| ">>"                                 { RSHIFT     }
| '^'                                  { EXPONENT   }
| '%'                                  { MODULO     }
| '*'                                  { MULTIPLY   }
| '/'                                  { DIVIDE     }
| '+'                                  { PLUS       }
| '-'                                  { MINUS      }
| '|'                                  { UNION      }
| '&'                                  { INTERSECT }
| '~'                                  { SETDIFF    }
| '>'                                  { GT         }
| ">="                                 { GTE        }
| '<'                                  { LT         }
| "<="                                 { LTE        }
| "=="                                 { EQUAL      }
| "!="                                 { NOT_EQUAL }
| "AND"                                { AND        }
| "OR"                                 { OR         }
| '='                                  { ASSIGN     }
(*                 Datatypes                 *)
| "void"                               { VOID       }
| "bool"                               { BOOL       }
| "int"                                { INT        }
| "float"                              { FLOAT      }
| "string"                             { STRING     }
| "matrix"                             { MATRIX     }
| "graph"                              { GRAPH      }
| "numset"                             { NUMSET     }
| "strset"                             { STRSET     }
| "struct"                              { STRUCT    }
(*                 Keywords                   *)
| "def"                                 { FUNC      }
| "link"                                { LINK      }
| "if"                                  { IF        }
| "else"                                { ELSE      }
| "loop"                                { LOOP      }
| "break"                               { BREAK     }
| "continue"                            { CONTINUE  }
| "return"                              { RETURN    }
| "case"                                { CASE      }
| "default"                             { DEFAULT   }
| "const"                               { CONST     }
(*                 Literals                   *)
| "true"                                { TRUE      }
| "false"                               { FALSE     }
| "NULL"                                { NULL      }
| ['-']?digit+ as lxm                   { INTLIT(int_of_string lxm)}
| ['-']?digit+['.']digit+ as lxm        { FLOATLIT(float_of_string lxm)}
| '"' (([^ '"'] | "\\\"")* as lxm) '"' { STRLIT(lxm) }
| ['a'-'z' 'A'-'Z']['a'-'z' 'A'-'Z' '0'-'9' '_']*  as lxm { ID(lxm) }
| ['0'-'9' '_']['a'-'z' 'A'-'Z' '0'-'9' '_']*      as lxm { syntax_error
(lexbuf) ("Invalid identifier name " ^ lxm) (!linecounter) }
| eof                                   { EOF }
(* Raising error for unidentified character*)
| _                                     { syntax_error (lexbuf) ("Unexpected
character detected") (!linecounter) }

and ml_comment = parse
 "*$"                                  { token lexbuf }
| "\n"                                  { incr linecounter ; ml_comment lexbuf }
| eof                                   { syntax_error (lexbuf) ("Expected '*$'
before EOF") (!linecounter) }
| _                                     { ml_comment lexbuf }

and sl_comment = parse
 "\n"                                  { incr linecounter ; token lexbuf }
| eof                                   { EOF }
| _                                     { sl_comment lexbuf }
```

2. parser.mly

```
%{ open Ast
let parse_error s =
 begin
   try
     let start_pos = Parsing.symbol_start_pos ()
     and end_pos = Parsing.symbol_end_pos () in
     Printf.printf "File \"%s\", line %d, characters %d-%d: \n"
       start_pos.pos_fname
       start_pos.pos_lnum
       (start_pos.pos_cnum - start_pos.pos_bol)
       (end_pos.pos_cnum - start_pos.pos_bol)
   with Invalid_argument(_) -> ()
 end;
 Printf.printf "Syntax error: %s\n" s;
 raise Parsing.Parse_error
%}

%token SEMICOLON COLON LPAREN RPAREN LBRACE RBRACE LBRACK RBRACK COMMA DOT
%token PLUS MINUS MULTIPLY DIVIDE MODULO EXPONENT INCR DECR LSHIFT RSHIFT UNION
INTERSECT SETDIFF
%token ASSIGN EQUAL NOT_EQUAL GT GTE LT LTE
%token IF ELSE LOOP RETURN LINK BREAK CONTINUE CASE DEFAULT CONST STATIC RENAME
%token INT FLOAT STRING BOOL VOID MATRIX GRAPH NUMSET STRSET STRUCT FUNC
%token AND OR NOT TRUE FALSE NULL
%token <string> ID TAG_BEGIN TAG_END
%token <string> STRLIT
%token <int> INTLIT
%token <float> FLOATLIT
%token EOF

%nonassoc IF
%nonassoc ELSE
%right ASSIGN
%left OR
%left AND
%left EQUAL NOT_EQUAL
%left GT GTE LT LTE
```

```
%left PLUS MINUS
%left MULTIPLY DIVIDE MODULO
%right NOT NEG
%nonassoc LPAREN RPAREN

%start start
%type <Ast.program> start

%%

start:
 decls EOF                      { $1 }

decls:  /* nothing */
   {{  globals = [];
       funcs = [];
       structs = [];
   }}
 | decls vdecl
   {{
     globals = $2 :: $1.globals;
     funcs = $1.funcs;
     structs = $1.structs;
   }}
 | decls fdecl
   {{
     globals = $1.globals;
     funcs = $2 :: $1.funcs;
     structs = $1.structs;
   }}
 | decls structdecl
   {{
     globals = $1.globals;
     funcs = $1.funcs;
     structs = List.rev ($2 :: (List.rev ($1.structs)));
   }}

fdecl: FUNC cmpd_typ ID LPAREN args RPAREN LBRACE var_decl_list stmt_list
RBRACE
 {{
   ret_type = $2;
   fname = $3;
   args = List.rev $5;
   local_vars = List.rev $8;
   body = List.rev $9;
 }}

structdecl:
 STRUCT ID LBRACE var_decl_list RBRACE
 {{
   name = $2;
   members = List.rev $4;
 }}

args: /* nothing */           { []        }
 | arg                       { [$1]      }
 | args COMMA arg            { $3 :: $1 }

arg:
   sc_specifier cmpd_typ ID             { check_primitive $1 $2 $3 Noexpr
}
 | sc_specifier cmpd_typ ID typ_size    { check_non_primitive $1 $2 $3 $4
Noexpr }

/* ----------------- Parsing Datatypes  ----------------- */
cmpd_typ:
   typ                       { Datatype($1) }
 | STRUCT ID                 { Struct($2)    }

/* keep typ uniform for both primitive and non-primitive */
typ:
   INT                       { Int       }
 | FLOAT                     { Float     }
 | STRING                    { String    }
 | BOOL                      { Bool      }
 | VOID                      { Void      }
 | MATRIX                    { Matrix    }

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

```
vdecl:
 /* declaration without definition */
   sc_specifier typ ID SEMICOLON                { check_primitive $1
(Datatype($2)) $3 Noexpr    }
 | sc_specifier typ ID typ_size SEMICOLON       { check_non_primitive $1
(Datatype($2)) $3 $4 Noexpr }
 | STRUCT ID ID SEMICOLON                       { check_primitive Normal
(Struct($2)) $3 Noexpr  }
 /* declaration with definition */
 | sc_specifier typ ID ASSIGN expr SEMICOLON    { check_primitive $1
(Datatype($2)) $3 $5        }
 | sc_specifier typ ID typ_size ASSIGN expr SEMICOLON
                                                { check_non_primitive $1
(Datatype($2)) $3 $4 $6 }

/* TODO: extend for set size, num nodes in graph... */
typ_size:
 LT INTLIT COMMA INTLIT GT                      { ($2, $4) }

sc_specifier: /* nothing */                      { Normal }
 | CONST                                        { Const  }
 | STATIC                                       { Static }
 | RENAME                                       { Rename }

/* ------------- Statements and Expressions ------------- */
stmt: expr SEMICOLON                             { Expr $1         }
 | BREAK SEMICOLON                              { Break Noexpr    }
 | CONTINUE SEMICOLON                           { Continue Noexpr }
 | RETURN expr SEMICOLON                        { Return $2 }
 | RETURN SEMICOLON                             { Return Noexpr }
 | LBRACE stmt_list RBRACE                      { Block(List.rev $2) }
 | IF LPAREN expr RPAREN stmt ELSE stmt         { If($3,$5,$7) }
 | IF LPAREN expr RPAREN stmt                   { If($3,$5,Block([]))}
 | LOOP LPAREN expr SEMICOLON expr_opt RPAREN stmt  { Loop($3,$5,$7)}
 | LOOP LPAREN expr RPAREN stmt                 { Loop($3,Null,$5)}

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

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

/* TODO: update other operators and unary minus, etc. */
expr:   INTLIT                                          { Intlit($1) }
 | FLOATLIT                                            { Floatlit($1) }
 | TRUE                                                { True  }
 | FALSE                                               { False }
 | NULL                                                { Null  }
 | ID                                                  { Id($1) }
 | STRLIT                                              { Strlit($1) }
 | LPAREN expr RPAREN                                  { $2 }
 | expr EQUAL expr                                     { Binop($1, Equal, $3) }
 | expr NOT_EQUAL expr                                 { Binop($1, Not_equal, $3) }
 | expr LT expr                                        { Binop($1, Lt, $3)  }
 | expr LTE expr                                       { Binop($1, Lte, $3) }
 | expr GT expr                                        { Binop($1, Gt, $3)  }
 | expr GTE expr                                       { Binop($1, Gte, $3) }
 | expr AND expr                                       { Binop($1, And, $3) }
 | expr OR expr                                        { Binop($1, Or, $3)  }
 | expr PLUS expr                                      { Binop($1, Add, $3) }
 | expr MINUS expr                                     { Binop($1, Sub, $3) }
 | expr MULTIPLY expr                                  { Binop($1, Mul, $3) }
 | expr DIVIDE expr                                    { Binop($1, Div, $3) }
 | expr MODULO expr                                    { Binop($1, Mod, $3) }
 | MINUS expr %prec NEG                                { Unop(Neg, $2)      }
 | NOT expr                                            { Unop(Not,$2)       }
 | LBRACK matrix_lit RBRACK                            { MatrixLit($2) }
 | ID LBRACK expr RBRACK                               { MatrixRow($1, $3) }
 | ID LBRACK expr COMMA expr RBRACK                    { MatrixElem($1, $3, $5) }
 | ID LBRACK expr COMMA expr RBRACK ASSIGN expr { MatrixModify($1, ($3,$5), $8)
}
 | ID DOT ID                                           { StructAccess($1, $3) }
 | ID DOT ID ASSIGN expr                               { StructAsgn($1, $3, $5) }
 | ID DOT ID LBRACK expr COMMA expr RBRACK             { StructMatrixAccess($1, $3,
($5, $7)) }
 | ID DOT ID LBRACK expr COMMA expr RBRACK ASSIGN expr { StructMatrixModify($1,
$3, ($5, $7), $10) }
 | ID LPAREN args_rev RPAREN                           { FunCall($1, $3) }
 | ID ASSIGN expr                                       { Asgn($1,$3)}


/* function call arguments */
args_rev:
   /* nothing */          { []       }  /* when function has no args */
 | args_list              { List.rev $1 }

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

/* rules for syntax to define a matrix */
matrix_lit:
   matrix_lit_row         { [|$1|]   }
 | matrix_lit_part        { $1       }

matrix_primitive:
   FLOATLIT               { $1       }
 | INTLIT                 { float_of_int $1 }
 | MINUS FLOATLIT         { -. $2    }
 | MINUS INTLIT           { float_of_int (-$2) }

matrix_lit_row:
   matrix_primitive       { [|$1|]   }
 | matrix_lit_row COMMA matrix_primitive { Array.append $1 [|$3|] }

matrix_lit_part:
 | LBRACK matrix_lit_row RBRACK COMMA LBRACK matrix_lit_row RBRACK {  [|$2; $6|]
}
 | matrix_lit_part COMMA LBRACK matrix_lit_row RBRACK { Array.append $1 [|$4|]
}
```

3. ast.ml

```
type operator = Add | Sub | Mul | Div | Mod | Equal | Not_equal | Lt | Lte | Gt
| Gte | And | Or
type datatype = Int | Bool | String | Float | Void | Matrix
type cmpd_typ = (* uniform for structs and other datatypes *)
```

```
   Datatype of datatype
 | Struct of string
type storage_class = Const | Static | Rename | Noexpr | Normal
type unary_operator = Not  | Neg

type expr =
   Binop of expr * operator * expr
 | Unop of unary_operator * expr
 | Intlit of int
 | Strlit of string
 | Floatlit of float
 | True
 | False
 | Id of string
 | Asgn of string * expr
 | FunCall of string * expr list
 | MatrixLit of float array array
 | MatrixRow of string * expr
 | MatrixElem of string * expr * expr
 | MatrixModify of string * (expr * expr) * expr
 | StructAccess of (string * string)
 | StructAsgn of (string * string * expr)
 | StructMatrixAccess of (string * string * (expr * expr))
 | StructMatrixModify of (string * string * (expr * expr) * expr)
 | Null
 | Noexpr

(* storage class, var type, var name, dimensions, defn *)
type bind = storage_class * cmpd_typ * string * (int * int) * expr

type statement =
   Block of statement list
 | Expr of expr
 | Break of expr
 | Continue of expr
 | Return of expr
 | If of expr * statement * statement
 | Loop of expr * expr * statement
 type fdecl = {
 ret_type : cmpd_typ;
 fname : string;
 args : bind list;
 local_vars: bind list;
 body : statement list;
}

type struct_decl = {
 name : string;
 members : bind list;
}

type program = {
 globals : bind list;
 funcs : fdecl list;
 structs : struct_decl list;
}

(* functions for datatype uniformity in parser *)

let check_non_primitive sc_specifier cmpd_typ variable_name typ_size expr =
 match cmpd_typ with
   Datatype a ->
   (
     match a with
         Matrix -> (sc_specifier, cmpd_typ, variable_name, typ_size, expr)
       | _ -> failwith("Primitive type, only non-primitive types can have a
size")
   )
   | Struct a -> failwith("Cannot have a size")

let check_primitive sc_specifier cmpd_typ variable_name expr =
 match cmpd_typ with
   Datatype a ->
   (
     match a with
       Matrix -> failwith("Non-primitive type: Must have size.")
     | _ -> (sc_specifier, cmpd_typ, variable_name, (-1, -1), expr)
   )
   | _ -> (sc_specifier, cmpd_typ, variable_name, (-1, -1), expr)
```

```
(*priting functions*)

let print_oper = function
   Add -> "+"
 | Sub -> "-"
 | Mul -> "*"
 | Div -> "/"
 | Equal -> "=="
 | Not_equal -> "!="
 | Lt -> "<"
 | Lte -> "<="
 | Gt -> ">"
 | Gte -> ">="
 | And -> "AND"
 | Or -> "OR"
 | Mod -> "%"


let print_uoper = function
   Neg -> "-"
 | Not -> "!"

let print_storage_class = function
   | Const -> "const"
   | Static -> "static"
   | Rename -> "rename"
   | Noexpr -> " "
   | Normal -> ""

let rec print_expr_string = function
  Intlit(l) -> string_of_int l
 | Floatlit(l) -> string_of_float l
 | True -> "true"
 | False -> "false"
 | Strlit(l) -> "\"" ^ (String.escaped l) ^ "\""
 | MatrixLit(_) -> "matLit"
 | Id(s) -> s
 | Binop(e1, o, e2) ->print_expr_string e1 ^ " " ^ print_oper o ^ " " ^
print_expr_string e2
 | Unop(o, e) -> print_uoper o ^ print_expr_string e
 | Asgn(v, e) -> v ^ " = " ^ print_expr_string e
 | FunCall(f, el) -> f ^ "(" ^ String.concat ", " (List.map print_expr_string
el) ^ ")"
 | MatrixRow(s,e) -> s ^ ".row"
 | MatrixElem(s, r, c) -> s ^ "[" ^ print_expr_string r ^ "]" ^ "[" ^
print_expr_string c ^ "]"
 | MatrixModify(s, (e1,e2), e3) -> s ^ "[" ^ print_expr_string(e1) ^ ", " ^
print_expr_string(e2) ^ "]" ^ " = " ^ print_expr_string(e3)
 | StructAccess((s1,s2)) -> s1 ^ s2
 | StructAsgn((s1,s2,e)) -> s1 ^ "." ^ s2 ^ " = " ^ print_expr_string(e)
 | StructMatrixAccess(s1,s2,(e1,e2)) -> s1 ^ s2 ^ "[" ^ print_expr_string(e1) ^
", " ^ print_expr_string(e2) ^ "]"
 | StructMatrixModify((s1,s2,(e1,e2),e3)) -> s1 ^ s2 ^ "[" ^
print_expr_string(e1) ^ ", " ^ print_expr_string(e2) ^ "]" ^ " = " ^
print_expr_string(e3)
 | Noexpr -> ""
 | Null -> ""

let print_typ_str = function
  Void -> "void"
 | Int -> "int"
 | Bool -> "bool"
 | Float -> "float"
 | String -> "string"
 | Matrix -> "matrix"


let print_cmpdtyp_info = function
   Datatype(d) -> print_typ_str d
 | Struct(s) -> "struct" ^ s


let rec print_stmt_string = function
  Block(stmts) -> "{\n" ^ String.concat "" (List.map print_stmt_string stmts) ^
"}\n"
 | Expr(expr) -> print_expr_string expr ^ ";\n";
 | Return(expr) -> "return " ^ print_expr_string expr ^ ";\n";
 | If(e, s, Block([])) -> "if (" ^ print_expr_string e ^ ")\n" ^
print_stmt_string s
 | If(e, s1, s2) ->  "if (" ^ print_expr_string e ^ ")\n" ^  print_stmt_string
s1 ^ "else\n" ^ print_stmt_string s2
 | Loop(e1, e2, s) -> "loop (" ^ print_expr_string e1  ^ " ; " ^
print_expr_string e2 ^ " ; " ^ ") " ^ print_stmt_string s
 | Break(e) -> "break" ^ print_expr_string e
 | Continue(e) -> "continue" ^ print_expr_string e

let string_of_vdecl (t, id) = print_cmpdtyp_info t ^ " " ^ id ^ ";\n"
let string_of_vdecl (s,t, id,_,expr) = print_storage_class s ^
print_cmpdtyp_info t ^ " " ^ id ^ " " ^ print_expr_string expr ^ ";\n"

 let string_of_fdecl fdecl =
   let thirdoffive = fun (_,_,y,_, _) -> y in
   print_cmpdtyp_info fdecl.ret_type ^ " " ^
   fdecl.fname ^ "(" ^ String.concat ", " (List.map thirdoffive fdecl.args) ^
   ")\n{\n" ^
   String.concat "" (List.map string_of_vdecl fdecl.local_vars) ^
   String.concat "" (List.map print_stmt_string fdecl.body) ^
   "}\n"
  let string_of_struct_decl s =
   let vdecls = String.concat "" (List.map string_of_vdecl s.members) in
   "struct " ^ s.name ^ "{\n" ^
     vdecls ^
   "}\n"
    let print_program_string program =
     let vars = program.globals
     and funcs = program.funcs
     and structs = program.structs in
     String.concat "" (List.map string_of_vdecl vars) ^ "\n" ^
     String.concat "" (List.map string_of_struct_decl structs) ^ "\n" ^
     String.concat "\n" (List.map string_of_fdecl funcs)
```

4. semant.ml

```
(* Semantic checker for Tureasy *)

open Ast
(* open Sast *)
```

```
module StringMap = Map.Make(String)    (* Symbol table *)

(* First we declared some helper functions which will be used in semantic
checking*)

let check_for_void errormsg = function
   (_, Datatype(typ), name, _, _) when typ = Void -> raise (Failure (errormsg
name))
 | _ -> ()

let check_duplicate errormsg lst =
 let rec help_check = function
     elem1 :: elem2 :: _ when elem1 = elem2 -> raise (Failure (errormsg elem2))
   | _ :: rem -> help_check rem
   | [] -> ()
 in help_check (List.sort compare lst)

(* redundant, merge later *)
let thirdoffive = function
 (_,_,y,_, _) -> y
 let secondoffive = function
 (_,y,_,_, _) -> y

let rec contains z = function
   [] -> false
 | head :: tail -> if z = head then true else contains z tail

let type_match types = function Datatype(t) -> contains t (Array.to_list types)
     | _ -> false

let get_struct_name = function
struct_typ -> struct_typ.name

let check_for_empty_struct errormsg = function
 { name = n; members = []; } -> raise (Failure (errormsg n))
 | _ -> ()

let check_duplicate_fields errormsg = function
  { name = n; members = memberlst; }
  -> check_duplicate (fun field -> "Found duplicate field '" ^ field ^ "' in
struct declaration") (List.map thirdoffive memberlst)

let rec check_assign_stmt lval rval =
 match (lval, rval) with
     (Datatype(p1),Datatype(p2)) -> if (p1 = p2 || (p1 = Int && p2 = Float) ||
(p1 = Float && p2 = Int)) then true else false
   | (Struct(s1), Struct(s2)) -> if s1 = s2 then true else
       (print_endline (s1 ^ s2); false)
   | _ -> false

let check_assign lval rval ex =
   if check_assign_stmt lval rval then lval
   else raise (Failure ("Illegal assignment of " ^ print_cmpdtyp_info lval ^
                 " = " ^ print_cmpdtyp_info rval ^ " in " ^
                 print_expr_string ex))

let get_struct_member_type struct_decl member errormsg =
 try
   let member_bind = List.find (fun (_,_,n,_, _) -> n = member)
struct_decl.members
   in secondoffive member_bind  (* return the typ *)
 with Not_found -> raise (Failure errormsg)



let semantic_check program =
  let global_vars = program.globals
 and funcs = program.funcs
 and structs = program.structs in
  (* CHECK FOR GLOBAL VARIABLES *)
 (* check for globals defined as void type *)
 List.iter (check_for_void (fun name -> "Found illegal void global '" ^ name ^
"'")) global_vars;
 (* check for duplicate variable names *)
 check_duplicate (fun name -> "Found duplicate global '" ^ name ^ "'")
(List.map thirdoffive global_vars);


 (* CHECK STRUCTS *)
 (* check for struct without any fields *)
 List.iter (check_for_empty_struct (fun name -> "Found struct without fields '"
^ name ^ "'")) structs;
 (* check for duplicate struct names *)
 check_duplicate (fun name -> "Found duplicate struct '" ^ name ^ "'")
(List.map get_struct_name structs);
 (* check for duplicate field names *)
 List.iter (check_duplicate_fields (fun fieldmessage -> fieldmessage)) structs;

(* above has been tested, and is working *)

 (* CHECK FOR FUNCTIONS *)
 let keywords_builtin = Array.to_list
   [|
     "print";
     (* **************more keywords to add*************** *)
   |]
 in
 (* if a builtin function's name has been used by user for some other
definition*)
 List.iter (fun fname ->
   if List.mem fname (List.map (fun fd -> fd.fname) funcs)
   then raise (Failure ("Function " ^ fname ^ " cannot be defined, it is
built-in"))
 ) keywords_builtin;
 (*check for duplicate function names as we don't support function
overloading*)
 check_duplicate (fun name -> "Duplicate function " ^ name )
   (List.map (fun fd -> fd.fname) funcs);


 let built_in_decls = StringMap.empty in
 (* Add all function names to symbol table *)
 let func_decls = List.fold_left (fun map fd -> StringMap.add fd.fname fd map)
                      built_in_decls funcs
 in
 (* check whether a function exists in symbol table *)
 let func_decl f = try StringMap.find f func_decls
     with Not_found -> raise (Failure ("Unknown function " ^ f ^ " call"))
 in
 (* main has to be declared in every program *)
 let _ = func_decl "main" in

 (* semantic check  inside each function *)
 let semantic_check_func func =

   List.iter (check_for_void (fun arg -> "Illegal void arguments " ^ arg ^
       " in " ^ func.fname)) func.args;

   check_duplicate (fun arg -> "Duplicate arguments " ^ arg ^ " in " ^
func.fname)
     (List.map thirdoffive func.args);

   List.iter (check_for_void (fun local -> "Illegal void locals " ^ local ^
       " in" ^ func.fname)) func.local_vars;

   check_duplicate (fun local -> "Duplicate locals " ^ local ^ " in " ^
func.fname)
     (List.map thirdoffive func.local_vars);

   (* Build local symbol table of variables for this function *)
   let symbols = List.fold_left (fun map (sc,t,n,(r,c),_) -> StringMap.add n t
map)
     StringMap.empty ( global_vars @ func.args @ func.local_vars )
   in

   (* Return a variable from our local symbol table if exists or throw error*)
   let identifier_type id =
     try StringMap.find id symbols
     with Not_found -> raise (Failure ("Identifier " ^ id ^ " is undeclared"))
   in

   let struct_decl = List.fold_left (fun map sd -> StringMap.add sd.name sd
map)
                         StringMap.empty structs
   in

   let get_struct_decl id =
     match identifier_type id with
     Struct name -> (
         try StringMap.find name struct_decl
       with Not_found -> raise (Failure ("Identifier " ^ id ^ " is
undeclared"))
     )
     | _ -> raise (Failure (id ^ "is not a struct " ))
   in

   (* semantic check of expressions inside functions*)
   let rec expr: expr-> cmpd_typ = function
       Intlit _ -> Datatype(Int)
     | Floatlit _ -> Datatype(Float)
     | True -> Datatype(Bool)
     | False -> Datatype(Bool)
     | Id id -> identifier_type id
     | Strlit _ -> Datatype(String)
     | Noexpr -> Datatype(Void)
     | Null -> Datatype(Void)
     | Asgn (var,e) as cmpd ->
       let left = identifier_type var
       and right = expr e in
       check_assign left right cmpd
     | MatrixLit m as m_expr -> let row_size = Array.length(m.(0)) in
                         let check_length l =
                           if Array.length(l) != row_size then
                             raise (Failure ("All rows must have same number of
elements in matrix literal: " ^ print_expr_string m_expr))
                         in
                         Array.iter check_length m;
                         Datatype(Matrix)
     | MatrixElem(m_id,ridx,cidx) as m_elem -> if (expr ridx) <> Datatype(Int)
then
                                                 raise (Failure ("Index of
matrix expected to be an integer, but found '" ^ print_expr_string ridx
                                                                 ^ "' which
has type  " ^ print_cmpdtyp_info (expr ridx)))
                                               else if (expr cidx) <>
Datatype(Int) then
                                                 raise (Failure ("Index of
matrix expected to be an integer, but found '" ^ print_expr_string ridx
```

```
                                                          ^ "' which has type  " ^
print_cmpdtyp_info (expr ridx)))
                                                       else if (identifier_type m_id)
<> Datatype(Matrix) then
                                                          raise (Failure ("Expected
matrix type, but found '" ^ m_id ^ "' which is declared as type "
                                                                         ^
print_cmpdtyp_info (identifier_type m_id)));
                                                       Datatype(Float)
      | MatrixModify(m_id,(ridx,cidx),a_ex) as m_ex ->  if (expr ridx) <>
Datatype(Int) then
                                                                   raise (Failure ("Index
of matrix expected to be an integer, but found '" ^ print_expr_string ridx
                                                                             ^ "'
which has type  " ^ print_cmpdtyp_info (expr ridx)))
                                                                 else if (expr cidx) <>
Datatype(Int) then
                                                                   raise (Failure ("Index
of matrix expected to be an integer, but found '" ^ print_expr_string ridx
                                                                   ^ "' which has type  "
^ print_cmpdtyp_info (expr ridx)))
                                                                 else if (identifier_type
m_id) <> Datatype(Matrix) then
                                                                   raise (Failure
("Expected matrix type, but found '" ^ m_id ^ "' which is declared as type "
                                                                             ^
print_cmpdtyp_info (identifier_type m_id)));
                                                                 check_assign
(Datatype(Float)) (expr a_ex) m_ex

      | StructAccess (name, member) -> ignore(identifier_type name); (*check
it's declared *)
          let s_decl = get_struct_decl name in (* get the ast struct_decl type
*)
          get_struct_member_type s_decl member
          ("Illegal struct member access: " ^ name  ^ "." ^ member)

      | StructAsgn (name, member, e) as ex ->  (* TODO: add illegal assign test
*)
          let t = expr e and struct_decl = get_struct_decl name in
```

```
         let member_t = get_struct_member_type struct_decl member
             ("Illegal struct member access: " ^ name  ^ "." ^ member) in
         check_assign member_t t ex

     | Binop (e1,op,e2) as ex ->
       let typ1 = expr e1 and
       typ2 = expr e2 in
       let res = match (typ1,typ2) with (Datatype(t1), Datatype(t2)) -> (
       let operand_type =
                     match op with
                         Add when (t1 = Int && t2 = Float) || (t1 = Float && t2
= Int) ||
                                  ((t1 = t2) && (t1 = Int || t1 = Float || t1 =
String )) -> t1
                       | Mul when (t1 = Int && t2 = Float) || (t1 = Float && t2
= Int) ||
                                  ((t1 = t2) && (t1 = Int || t1 = Float)) -> t1
                       | Sub | Div when (t1 = Int && t2 = Float) || (t1 = Float
&& t2 = Int) ||
                                        ((t1 = t2) && (t1 = Int || t1 = Float))
-> t1
                       | Mod when (t1 = t2) && (t1 = Int) -> t1
                       | Equal | Not_equal when (t1 = Int && t2 = Float) || (t1
= Float && t2 = Int) ||
                                                ((t1 = t2) && (t1 = Int || t1 =
Float || t1 = String)) -> Bool
                       | Lt | Lte | Gt | Gte when (t1 = Int && t2 = Float) ||
(t1 = Float && t2 = Int) ||
                                                  ((t1 = t2) && (t1 = Int || t1
= Float)) -> Bool
                       | And | Or when (t1 = t2) && (t1 = Bool) -> Bool
                       | _ -> raise (Failure ("Illegal binary operator " ^
print_cmpdtyp_info typl ^ " " ^
                                     print_oper op ^ " " ^ print_cmpdtyp_info
typ2 ^
                                     " usage in " ^ print_expr_string ex))

                         (* Matrix operations also need to be added here *)
                     in Datatype(operand_type)
           )
```

```
          | _ -> raise (Failure("Operator not found"))
      in res
    | Unop (op,e) as ex ->
      let typ = expr e in ( match typ with Datatype(dt) -> (

          let operand_type =
            match op with
            Neg when dt = Int || dt = Float -> dt
          | Not when dt = Bool -> Bool
          | _ -> raise (Failure ("Illegal unary operator " ^ print_uoper op ^
" usage in " ^ print_expr_string ex))

      in Datatype(operand_type)
      )
        | _ -> raise (Failure ("This operator has not been implemented")))

    (* Currently only print is the built in function,
    any other built in functions we would like to include must
    be first added to the function keywords_builtin and then
    corresponding Funcall must be added here  *)

    (* Example Funcall for print *)
    (*TODO: argument mismatch while var decls *)
    | FunCall ("print",_) -> Datatype(Int)
    | FunCall ("print",_) -> Datatype(String)
    | FunCall (fn,arg_list) as call ->
          let fd = func_decl fn in
          if List.length arg_list != List.length fd.args then
            raise (Failure ("Expecting " ^  string_of_int (List.length
fd.args) ^  " number of arguments in " ^ print_expr_string call))
          else
            List.iter2 (fun (sc,arg,name,_ ,_) e -> let typ = expr e in
                ignore (if check_assign_stmt arg typ then arg
                        else raise (Failure ("Illegal arguments found!
Expected " ^ print_cmpdtyp_info arg ^ " in " ^
                                            print_expr_string e ^ " but got " ^
print_cmpdtyp_info typ))))
            fd.args arg_list;
            fd.ret_type
  in
```

```
   let check_bool_expr b = if not (type_match [|Bool|] (expr b))
       then raise (Failure ("Expected boolen expression in " ^
print_expr_string b))
       else () in

   (* semantically-checking statement *)
   let rec semantic_check_stmt = function
       Block blk -> let rec check_block = function
         [Return _ as s] -> semantic_check_stmt s
       | Block blk :: blks -> check_block (blk @ blks)
       | s :: blks -> semantic_check_stmt s; check_block blks
       | [] -> ()
     in check_block blk
     | Expr e -> ignore (expr e)
     | Break e ->
           if e != Noexpr then raise (Failure ("Break statement should include
Noexpr, " ^
                                                    print_expr_string e ^ "
found"))
     | Continue e ->
           if e != Noexpr then raise (Failure ("Continue statement should
include Noexpr, " ^
                                                    print_expr_string e ^ "
found"))
     | Return r -> let rt_expr = (expr r) in
         if (check_assign_stmt rt_expr func.ret_type) then () else
         raise (Failure ("Return gives " ^ print_cmpdtyp_info rt_expr ^ " but
expected " ^
                       print_cmpdtyp_info func.ret_type ^ " in " ^
print_expr_string r ))
     | If (e,b1,b2) -> check_bool_expr e; semantic_check_stmt b1;
semantic_check_stmt b2;
     | Loop (e1,e2,st) -> check_bool_expr e1; ignore (expr e2);
semantic_check_stmt st;

   in
   semantic_check_stmt (Block func.body)
```

```
 in
 List.iter semantic_check_func funcs
```

5. codegen.ml

```
module L = Llvm
module A = Ast
module P = Printf
module StringMap = Map.Make(String)

let translate program =
 let globals = program.A.globals in
 let functions = program.A.funcs in
 let structs = program.A.structs in
 let context = L.global_context() in
 let the_module = L.create_module context "Tureasy" in


 let i32_t  = L.i32_type context in
 let i1_t   = L.i1_type context  in
 let i8_t   = L.i8_type context  in
 let float_t = L.double_type context in
 let string_t = L.pointer_type i8_t in
 let void_t = L.void_type context in
 let matrix_t = L.pointer_type i32_t in

 let datatype_infer = function
     A.Bool         -> i1_t
   | A.Void         -> void_t
   | A.Int          -> i32_t
   | A.Float        -> float_t
   | A.Matrix       -> matrix_t
   | A.String       -> string_t

 in

 let lcmpd_type_infer struct_decl_map = function
     A.Datatype (typ) -> datatype_infer (typ)
   | A.Struct (s)         -> L.pointer_type (fst (StringMap.find s
struct_decl_map))
```

```
 in

 let struct_decl_map =
   let add_struct m structdecl =
     let name = struct_decl.A.name
     and members = Array.of_list
       (List.map (fun (_,t, _,_,_) -> lcmpd_type_infer m t)
struct_decl.A.members) in
     let structname = L.named_struct_type context ("struct." ^ name) in
       L.struct_set_body structname members false;
           StringMap.add name (structname, structdecl) m in
   List.fold_left add_struct StringMap.empty structs in

 let struct_lltype_list =
   let bindings = StringMap.bindings struct_decl_map in
   List.map (fun (_, (typ_l, _)) -> L.pointer_type typ_l) bindings
 in

 let get_struct_pointer_lltype llval =
   let typ_l = L.type_of llval in
   A.get_try typ_l struct_lltype_list
 in

 let str_check_empty = L.define_global "__empty string" (L.const_string context
"") the_module in
 let initialize = function
   A.Datatype(typ) -> (
     match typ with
         A.Float -> L.const_float float_t 0.0
       | A.String -> L.const_bitcast str_check_empty string_t
       | A.Matrix -> L.const_null matrix_t
       | A.Bool  -> L.const_int i1_t 0
       | _ -> L.const_int i32_t 0
     )
   | A.Struct(_) as typ -> L.const_null (lcmpd_type_infer struct_decl_map typ)
 in
  let print_t = L.var_arg_function_type i32_t [| L.pointer_type i8_t |] in
 let print_func = L.declare_function "print" print_t the_module in
  let global_vars =
     let global_var m (typ,name) =
   let init  = initialize typ in
   let global_val_ll = L.define_global name init the_module in
   StringMap.add name (global_val_ll, typ) m in
   let globs = List.map (fun (_,t,n,_,_) -> (t,n)) globals in
   List.fold_left global_var StringMap.empty globs in

 let func_decls =
   let func_decl m fdecl =
   let name = fdecl.A.fname and
     args_type= Array.of_list
       (List.map(fun (_,t,_,_,_) -> lcmpd_type_infer struct_decl_map t)
fdecl.A.args) in
       let ftyp = L.function_type (lcmpd_type_infer struct_decl_map
fdecl.A.ret_type) args_type in
       StringMap.add name( L.declare_function name ftyp the_module,fdecl) m in
       List.fold_left func_decl StringMap.empty functions in

 let find_func fname = StringMap.find fname func_decls in

 let build_predefined fname actuals the_builder =
   let (fdef, fdecl) = (try StringMap.find fname func_decls with
       Not_found -> raise (Failure("Not defined: " ^ fname))) in
   let result = (match fdecl.A.ret_type with
       A.Datatype(t) when t = A.Void -> ""
       | _ -> fname ^ "_res")
 in
 L.build_call fdef actuals result the_builder in

 let build_func fdecl =
   let (func, _) = try StringMap.find fdecl.A.fname func_decls with Not_found
-> raise (Failure("Error 127")) in
   let builder = L.builder_at_end context (L.entry_block func) in

   let local_vars =
     let add_arg m (s, name) p = L.set_value_name name p;
     let locals = L.build_alloca (lcmpd_type_infer struct_decl_map s) name
builder in
     ignore (L.build_store p locals builder);
     StringMap.add name (locals, s) m
   in
```

```
     let add_local m (t, n) =
       let local_var = L.build_alloca (lcmpd_type_infer struct_decl_map t) n
builder in
       ignore (L.build_store (initialize t) local_var builder);
       StringMap.add n (local_var, t) m
       in

 let args = List.fold_left2 add_arg StringMap.empty (List.map (fun (_,t,n,_,_)
-> (t,n)) fdecl.A.args) (Array.to_list (L.params func)) in
 List.fold_left add_local args (List.map (fun (_,t,n,_,_) -> (t,n))
fdecl.A.local_vars) in

 let lookup_llval x = try fst (StringMap.find x local_vars)
                 with Not_found -> fst (StringMap.find x global_vars)
 in

 let lookup_type n = try snd (StringMap.find n local_vars)
                 with Not_found -> snd (StringMap.find n global_vars)
 in

 let get_struct_decl sname =
   try
     let typ = lookup_type sname in
     match typ with
       A.Struct(s) -> snd (StringMap.find s struct_decl_map)
     | _ -> raise Not_found
   with Not_found -> raise (Failure(sname ^ " not declared"))
 in

 let int_ops = function
     A.Add     -> L.build_add
   | A.Sub     -> L.build_sub
   | A.Mul    -> L.build_mul
   | A.Div     -> L.build_sdiv
   | A.Mod     -> L.build_srem
   | A.Or      -> L.build_or
   | A.And     -> L.build_and
   | A.Equal   -> L.build_icmp L.Icmp.Eq
   | A.Not_equal     -> L.build_icmp L.Icmp.Ne
   | A.Gt -> L.build_icmp L.Icmp.Sgt
   | A.Gte     -> L.build_icmp L.Icmp.Sge
   | A.Lt    -> L.build_icmp L.Icmp.Slt
   | A.Lte     -> L.build_icmp L.Icmp.Sle
   in

  let float_ops = function
     A.Add     -> L.build_fadd
   | A.Sub     -> L.build_fsub
   | A.Mul    -> L.build_fmul
   | A.Div     -> L.build_fdiv
   | A.Equal   -> L.build_fcmp L.Fcmp.Ueq
   | A.Not_equal     -> L.build_fcmp L.Fcmp.Une
   | A.Lt    -> L.build_fcmp L.Fcmp.Ult
   | A.Lte     -> L.build_fcmp L.Fcmp.Ule
   | A.Gt -> L.build_fcmp L.Fcmp.Ugt
   | A.Gte     -> L.build_fcmp L.Fcmp.Uge
   | _         -> raise Not_found
   in

  let bool_ops = function
     A.And       -> L.build_and
   | A.Or        -> L.build_or
   | _           -> raise Not_found
  in

 let rec expr builder = function
     A.Intlit i       -> L.const_int i32_t i
   | A.Floatlit f     -> L.const_float float_t f
   | A.True           -> L.const_int i1_t (1)
   | A.False          -> L.const_int i1_t (0)
   | A.Strlit s       -> L.build_global_stringptr (Scanf.unescaped s) "strlit"
builder
   | A.Noexpr         -> L.const_int i32_t 0
   | A.Null           -> L.const_pointer_null void_t
   | A.Id id          -> let var = L.build_load (lookup_llval id) id builder
                         in var
   | A.Asgn (var,e)   -> let e1 = expr builder e
                    in ignore (L.build_store e1 (lookup_llval var) builder); e1
   | A.Unop (op,e)      ->
```

```
       let e1 = expr builder e in
       let typ = L.type_of e1 in
       (match op with
           A.Neg ->
             if  typ = float_t then L.build_fneg e1 "f_neg" builder
             else L.build_neg e1 "neg" builder
         | A.Not -> L.build_not e1 "not" builder
       )
   | A.Binop (e1,op,e2)->
     let e11 = expr builder e1 and e22 = expr builder e2 in
     let typ1 = L.type_of e11 and typ2 = L.type_of e22 in
     let typs = (typ1, typ2) in
     (
       if typs = (i32_t, i32_t) && op = A.Mod then (build_predefined "mod"
[|e11; e22|] builder)
       else if typs = (i32_t, i32_t) then (int_ops op e11 e22 "int_ops"
builder)
       else if typs = (float_t, float_t) then (float_ops op e11 e22 "float_ops"
builder)
       else if typs = (i1_t, i1_t) then (bool_ops op e11 e22 "bool_ops"
builder)
       else raise (Failure ((A.print_oper op) ^ " not defined for "
                   ^ (L.string_of_lltype typ1) ^ " and " ^ (L.string_of_lltype
typ2) ^ " in "
                   ^ (A.print_expr_string e2)))

     )
   | A.FunCall ("print", arg) ->
     let args = List.map (expr builder) arg
     in L.build_call print_func (Array.of_list args) "print" builder
   | A.FunCall (fn,arg_list) ->
     let args = Array.of_list (List.rev (List.map(expr builder) (List.rev
arg_list))) in
     let (func_def, func_decl) = find_func fn
     in L.build_call func_def args fn builder
 in

 let add_terminal builder instr =
       match L.block_terminator (L.insertion_block builder) with
         Some _ -> ()
```

```
      | None -> ignore (instr builder)
in

let rec stmt builder = function
    A.Block blk        -> List.fold_left stmt builder blk
  | A.Expr e           -> ignore (expr builder e); builder
  (* | A.Break e          -> ignore (L.build_br builder);
  | A.Continue e       -> ignore (L.build_br builder); *)
  | A.If (e,b1,b2)     ->
    let condition = expr builder e in
    let then_blk = L.append_block context "then" func in
    let merge_blk = L.append_block context "merge" func in
    add_terminal (stmt (L.builder_at_end context then_blk) b1)
    (L.build_br merge_blk);

    let else_blk = L.append_block context "else" func in
    add_terminal (stmt (L.builder_at_end context else_blk) b2)
    (L.build_br merge_blk);

    ignore (L.build_cond_br condition then_blk else_blk builder);
    L.builder_at_end context merge_blk

  | A.Return r         ->
    ignore (match fdecl.A.ret_type with
      A.Datatype (typ) when typ = A.Void -> L.build_ret_void builder
      | _ -> L.build_ret (expr builder r) builder ); builder

  | A.Loop (e1,e2,st)  ->

    let body = A.Block [st;A.Expr e2] in   (*thoda doubt*)
    let body_blk = L.append_block context "loop_body" func in
    let pred_blk = L.append_block context "loop" func in
             ignore (L.build_br pred_blk builder);
    add_terminal (stmt (L.builder_at_end context body_blk) body)
    (L.build_br pred_blk);
    let merge_blk = L.append_block context "merge" func in
    let pred_builder = L.builder_at_end context pred_blk in
    let condition = expr pred_builder e1 in
    ignore (L.build_cond_br condition body_blk merge_blk pred_builder);
    L.builder_at_end context merge_blk
```

```
in

let builder = stmt builder (A.Block fdecl.A.body)
in

add_terminal builder (match fdecl.A.ret_type with

  A.Datatype(typ) when typ = A.Float -> L.build_ret (L.const_float float_t
0.0)
  | A.Datatype(typ) when typ = A.Void -> L.build_ret_void
  | typ -> L.build_ret (L.const_int (lcmpd_type_infer struct_decl_map typ) 0)
)
in
(* ending the functions conversion*)
List.iter build_func functions;
the_module
```

7.3 Tests

7.3.1 - fail_args.tz

```
def int Triple_sum(int a, int b, int c)
{
    return a+b+c;
}

def int main()
{
    int x=10;
    int y=100;
    int z=-10;

    int res;
    res = Triple_sum(x,y);

    return 0;
}
```

7.3.2 - fail_binop.tz

```
def void foo (int a)
{
    int x = a;
    x = x+ "b";
    print(x);
    return;
}
def void main()
{
    int a=10;
    print("HI");
    foo(a);
}
```

7.3.3 - fail_builtincall.tz

```
def void print (int a)
{
    int x;
    return 0;
}

def int main()
{
    print(10);
}
```

7.3.4 - fail_dup_var.tz

```
def int main(int a)
{
    int x1 = 5;
    int x1 = 10;
    return 0;
}
```

7.3.5 - fail_dupfield.tz

```
int a;

struct x {
    int c;
    float c;
}

def void main ()
{
    int b;

    return;
}
```

7.3.6 - fail_dupglobalvar.tz

```
int a;
float a;

def void main ()
{
    int b;

    return;
}
```

7.3.7 - fail_emptystruct.tz

```
int a;

struct x {

}

def void main ()
{
    int b;
    return;
}
```

7.3.8 - fail_illegal_assignment_struct.tz

```
struct dem{
    int i;
    float j;

}

def void main()
{
    struct dem d;
    d.i="hello";
    return ;
}
```

7.3.9 - fail_matrixaccess.tz

```
def void main()
{
    matrix m<3,3>;
    string n;
    m = [[1.0,0,0],[0,1,0],[0,0,1]];
    n[1,2] = 1.5;

    return;
}
```

7.3.10 - fail_matrixliteral.tz

```
def void main()
{
    matrix m<3,3>;
    m = [[1.0,0,0],[0,1,0],[0,0]];

    return;
}
```

7.3.11 - fail_mlcomment.tz

```
$*
def void main()
{
   return;
}
```

7.3.12 - fail_no_main.tz

```
def int incr(int a)
{
    return a+1;
}
```

7.3.13 - pass_emptymain.tzpass_globaldecl.tz

```
def void main()
{
   return;
}
```

7.3.14 - pass_int_float_comp.tz

```
def void main()
{
      int a=1;
      float b=0.1;
      a=a+b;
}
```

7.3.15 - pass_int_float_comp_struct.tz

```
def void main()
struct dem{
    int i;
    float j;
}

def void main()
{
    struct dem d;
    d.j=2;
    return ;
}
```

7.3.16 - test1.tz

```
def void fun(int x)
{
      return x;
}
```

7.3.17 - test2.tz

```
def void main()
{
      int x;
      int x;
}
```