COMS W4115 Fall 2011 Course Project TML Final Report

Jiabin Hu (jh3240)
Akash Sharma (as4122)
Shuai Sun (ss4088)
Yan Zou (yz2437)

Columbia University

December 22nd, 2011

Contents

1 Introduction

1.1 Motivation

Tree is one of the most fundamental data structures not only in computer science, but also in real life. The applications built on it range from data storing and searching to coding and routing algorithms. However, in most modern programming languages, representing a tree requires pointers or references, which often leads to bugs that are hard to catch. Furthermore, codes on tree manipulation are usually difficult to read, since they hardly reflect the abstracted operation. Therefore, we plan to design a new language, the Tree Manipulating Language (TML), specifically for manipulations on trees. The goal of the language is to provide more efficient and user-friendly programming methods to implement operations on trees.

1.2 Overview

Tree manipulation language (TML) is a user friendly language that is designed to help users program trees. It allows users to create, manipulate and run algorithms on trees. Various existing programming languages make tree operations cumbersome and TML is intended to bridge this gap. In TML everything is of type Tree except primitive data types. Each node of the tree is of type Tree and it has fields associated with it like parent, child etc. Every node of the Tree is root for its subtree. Nodes of the tree can consist of user defined or primitive value types and they can be of any degree. We can refer to any child or parent node of a given node using predefined constructs. TML provides methods to perform trivial operations on trees such as tree traversal, node creation or deletion etc.

In TML, we introduce a new type named type ***Tree***. As our language is specifically designed for tree programs, incurring a type ***Tree*** will make it easier to program. Basic operations to program on a tree are provided in our language, such as tree construction, adding tree node, referring to father, referring to root data, etc. Programmers could both use the provided operation or define new functions to manipulate trees.

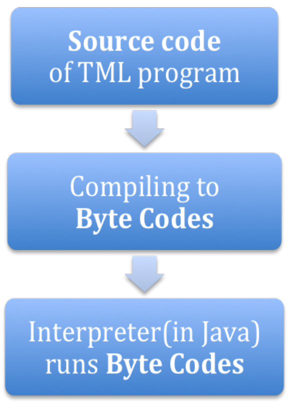

Figure 1: TML Compiling Process

The main concept of our language is that, everything except primitive types is regarded as a tree, just like everything in Java is an object. Noted that every child of a tree is the root of its sub-tree. In TML, we regard all nodes in a tree as sub trees which are of the same type as the original one. When applying operations on a tree, we recursively apply the operations on sub trees and the root. The recursive feature of trees is the reason for this language feature. For operations on a single node, reference to the node is available by referring to its sub tree.

In TML, users can define new types of tree inherited from the basic type ***Tree***. The degree and storage field can also be user-defined. Users could use this feature to build their own trees and even queues, stacks, lists, etc.

TML compiles the source codes and translate them into TML byte code, which is then interpreted by TML interpreter to execute the program. The compiler and byte code generator is implemented in OCaml, and the interpreter is written in Java Figure 1 shows the translation process of TML.

2 Language Tutorial

TML was designed to describe trees and use them in a more convenient way, with language-supported tree operators. User of TML can use these features to define their own tree-like data structures and write their programs.

2.1 An Introduction

TML is a C-like language. Most of the grammar is similar to that of C. Each source file, with the extension ".tml", should contain a main() method, which returns nothing. And each statement in the code should end with a semicolon.

```
void main ()
{
    // tml statements.
}
```

TML includes primitive types, which are integer, float, character, boolean, and string. These types are used in the way just like in C and C++, only that in TML, no type conversion is allowed. Also, user can define their own types of tree in TML. The definition of the tree type should appear before they are used in other parts of the program. Below is an example of user defined binary tree.

```
treetype <2, [left, right]>MyTree_t
{
    int vint = 0;
    float vflt = 1.;
    string vstr = "hi";
}
```

User can declare new instances of tree types in the same way in C. To allocate memory space of an instance, user can use the built-in alloc function. TML programs can output results of computation by using built-in printing function.

```
MyTree_t ta;        // declare new instance
alloc (ta);         // alloc memory
ta.vint = 15;
print (ta.vint);    // prints out 15.
```

When your finish writing the code, cd to the src directory in our project, and type the following command line to compile the program:

```
./tmlc sample_program.tml
```

To run the TML program, type the following command line:

```
./tml sample_program.tmb
```

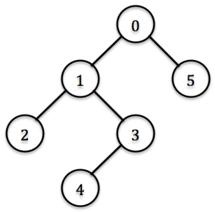

Figure 2: Tree construction in sample code

2.2 An Example of Tree Traversal

The following code of TML is an example of inorder traversal over the tree. In the program, it defineds a binary tree type, constructs a tree with tree operators, sets the value and prints the result. The tree constructed along with the value set is in the figure 2.

```
treetype <2, [left, right]>MyTree_t
{
    int val = 0;
}
void main ()
{
    int i = 0;
    MyTree_t a, b, c, d, e, f;
    alloc(a, b, c, d, e, f);
    a ->( b ->( d : e ->( f : ~ ) ) : c);
    foreach node in a by preorder
    {
        node.val = i;
        i = i + 1;
    }
    foreach node in a by inorder
    {
        print(node.val);
        print(" ");        // print a space
    }
}
```

Now, you have already written a program in TML. Let's name it tutorial.tml. You can compile it and run by the following command.

```
./tmlc tutorial.tml
./tml tutorial.tmb
```

It should prints out:

```
2 1 4 3 0 5
```

3 Language Manual

3.1 Lexical Conventions

There are different classes of tokens that are supported in TML. Token types are identifiers, keywords, literals, strings and operators. As in C language, whitespace characters are ignored except insofar as they serve to delineate other tokens in the input stream. If the input stream has been parsed into tokens up to a given character,the next token is taken to include the longest string of characters which could possibly constitute a token.

3.1.1 Character Set

TML takes the standard **ASCII** character set for source codes.

3.1.2 Comments

In TML, there are two ways to make comments. The first style starts with the characters /*, and ends with the characters */, all contents between which are comments. Note that, just like C-style comments, TML supports only un-nested comments. The second style is inline comments. It starts with the characters //, all contents in the current line after which are regarded as comments.

```
/* This is a
   block comment. */
// This is an inline comment.
```

3.1.3 Identifiers

In TML, an identifier is a string that starts with a letter or an underscore, and consists of a sequece of letters, digits, and underscores. All identifiers are case-sensitive in TML.

3.1.4 Keywords

In TML, the words listed in Table 1 are reserved as keywords, and are not allowed to be used as a user-defined identifier.

if	else	do	while	for
break	continue	foreach	in	by
return	void	main	print	alloc
preorder	inorder	postorder	levelorder	treetype
int	float	char	string	bool
true	false			

Table 1: Keywords in TML

3.1.5 Constants

There are several kinds of constants in TML, which are listed as follows.

- Integer constants An integer constant consists of a sequence of digits, starting with a non-zero digit. All integer constants are considered to be decimal and of type integer.

- Float constants A floating constant consists of an integer part, a decimal point, a fraction part, and optionally an 'e' with a signed integer exponent. The integer and fraction parts both consist of a sequence of digits, and either one could be missing, but not both. Either the decimal point or the 'e' with the exponent could be missing, but not both. All floating constants are of type float.

- Character constants A character constant is a single ASCII character enclosed by $'$ $'$, which is of character type. Note that $'\backslash n'$, $'\backslash t'$, and $'\backslash r'$, are character constants.

- String constants A string constant consists of several character constants enclosed by $''$ $''$, and implicitly ends with $'\backslash 0'$.

- Boolean constants There are only two boolean constants, *true* and *false*.

- Tree constant There is only one tree constant, which is the null constant. The null constant is a null value of any tree type. If a tree is null, then it means the tree reference is lost at the point of time. In TML, the null constant is written as a tilde, '$\sim$'.

3.2 Types

In TML, there are two kind of types, the *primitive types* and the *tree types*.

3.2.1 Primitive Types

There are several kinds of primitive types. The size and value range of each type is listed in Table 2. The type specifiers can be found in section 3.4.1

Primitive Type	**Size**	**Range**	**Default value**
Integer	4 bytes	-2 147 483 648 2 147 483 647	0
Float	4 bytes	about ±3.402 823 47E+38	0.0
Character	2 byte	ASCII	$'\backslash 0'$
String	$\geq$ 1 byte	combinations of characters	$''''$*(empty)*
Boolean	4 bytes	true, false	false

Table 2: Primitive Types in TML

- Intergers In TML, there is only one type of integer. The integers are signed, and are of fixed size. The size of an integer is four bytes.

- Floating point numbers In TML, there is only one type of floating point number. The size of the float type is four bytes.

- Characters In TML, there is only one type of character. The size of a character is two bytes. The characters are interpreted as ASCII code.

- Strings Strings are considered as a primitive type in TML. A string is a permutation of more than one characters. A string can be compared with another string. The size of a string equals to the number of characters in it.

- Booleans Boolean type is a primitive type in TML. There are only two values, true and false, in the Boolean type, which is used to determinate logic conditions. There is no conversions between integers and these two values.

3.2.2 Tree Types

In TML, a *tree type* is a type to represent trees. By using the tree types, users can define and use tree data structures in their programs.

In a tree data structure, children of the root node can be also regarded as roots of sub trees. By this concept, in TML, all nodes in a tree are of the same tree type with the root. There is no tree node type in TML.

- Define a tree type Tree types are defined by users before they can declare and define variables of the tree types. A tree type consists of necessary parts, *type name*, *degree* and *member variable(s)*, and an optional part, *children index aliases*. To define a tree type, the *type name*, *degree* and *member variable(s)* must be defined, and the *children index aliases* can be optionally defined. The size of a tree type is the sum size of its member variable types.

  When defining each member variable, an initial value can be optimally defined. If the initial value of a member variable is not defined, it will be initialized with the default value of each type. For primitive types, the default values are defined in Table 2. For tree types, it will be assigned with null.

  An example of a tree type definition, MyTree_t, is given below.

  ```
  treetype <2, [left, right]>MyTree_t
  {
      int vint = 0;
      float vflt = 1.;
      string vstr = "hi ";
  }
  ```

- Type name The *type name* of a tree type follows the identifier definition in the section 3.1.3. After the tree type is defined, the type name can be used to define variables of this type.

- Degree In TML, each tree type must define a fixed *degree* at definition. All nodes of the same tree type have the same degree. The value range of the degree is from 1 to 99. The degree of a tree type can be referred to by using the operator "&" introduced in the section 3.3.6.

- Member variables In TML, a tree type can define its own *member variable(s)* to store values for algorithms and programs. Each tree type can have at most 99 member variables. Each member variable must have a name which is unique in its tree type, and can be of any primitive types or tree types. Note that if a member variable is defined as some tree type, the definition of the tree type must appear before the member variable definition. All member variables are public. The member variables of a tree type variable can be referred to by using dot operator introduced in the section 3.3.6.

- Children index and aliases In TML, children sub roots of a root can be referred to by the index number, ranging from 0 to (degree-1). Optionally, at the definition of a tree type, user can define alias of the children indices. The alias follows the identifier definition in section 3.1.3. Note that if aliases are defined, then aliases for all Children indices must be defined. Only defining a subset of children indices is not allowed in TML. Whether using number indices or aliases, the children of a tree root can be referred to by using child access operator introduced in section 3.3.6.

3.2.3 Type Conversions

In TML, no type conversion is allowed.

3.3 Expressions

In TML, expressions consist of operators and their operands. In this section, definition for each operator is given. To avoid ambiguity, precedence rules of the operators in TML are also defined in this section.

3.3.1 Arithmetic operators

In TML, arithmetic operators are $+$, $-$, $*$, $/$ and $\%$. $+$ means addition, $-$ means substruction, $*$ means multiplication, $/$ means division and $\%$ means modulation. All of them are binary and left associative. It requires that their operands must be of the same primitive types, and the result will be of the same type.

3.3.2 Comparative operators

In TML, comparative operators are $>$ (greater than), $<$ (less than), $>=$ (greater than or equal to), $<=$ (less than or equal to), $!=$ (not equal) and $==$ (equal). All of them are binary operators and left-associative. It requires that their operands must be of the same primitive types . The return value is a boolean value indicates the predicate.

3.3.3 Logical operators

Logical operators in TML include && (logical and), || (logical or) and ! (logical not). && and || are binary operators and left-associative. They take two operands of type boolean, and return a boolean value. ! is unary and appears on the left side of the operand. They type of the operand must be of type boolean and the return type is also a boolean value.

3.3.4 Assignment operators

TML's assignment operator is =. it's binary operator and right-associative. The left operand must be a legal left value, and the right operand must be an expression. In TML, a legal left value can be a variable, a member variable of a tree type or a child of a tree. When an assignment is taken, the value of the expression on the right is assigned to the left value, and the new value of the left value is returned, which allows chained assignments.

3.3.5 Parentheses

In TML, parentheses in expressions are used to overwrite the precedence rules. Expression enclosed by the parentheses is calculated before applying its adjacent operators.

3.3.6 Tree Operators

The TML is a set of operators that can be applied to trees. They are divided into three categories: tree-building operators, tree-copying operators and tree-querying operators.

- Tree-building operators The operator ->is the tree building operator. This operator is used to connect the root and its children to build a tree. It is binary and right-associative. Its left operand is a single tree node, representing the root. The right operand is a list of children trees, enclosed by parentheses. Each child tree is separated by colons, representing the immediate children of the root. An example is given below. In this example, a tree is built with ta as root and tb as its left subtree, and tc as its right subtree. This operator returns the newly built tree, which is ta.

  ```
  MyTree_t ta, tb, tc;
  alloc(ta, tb, tc);
  ta ->(tb : tc);
  ```

- Tree-copying operators In TML, two operators are supplied to copying trees. They are the At(@) operator and the Dollar($) operator. Both of them are right-associative. And they both take only one operand of a tree type.

  The At operator copies the root node of the operand and its values only, with all of its children set to null. The Dollar operator copies the whole tree referenced by the operand, including all sub trees and children. Both of them return the reference of the newly-copied tree.

  In the example below, ta_copy1 gets only the ta node with both children set null, while ta_copy2 get the copy of the whole ta tree, with the two children connected.

  ```
  MyTree_t ta, tb, tc;
  alloc(ta, tb, tc);
  ta ->(tb : tc);
  MyTree_t ta_copy1, ta_copy2;
  ta_copy1 = @ta;         //copy ta node only
  ta_copy2 = $ta;         //copy the whole ta tree
  ```

- Tree-querying operators In TML, several tree-querying operators are defined to get properties of a tree. They are the **Hash(#)** operator, the **square bracket operator([])**, the **Dot(.)** operator, the **Ampersand(&)** operator and the **Caret(ˆ)** operator. The detailed definition for each operator is given below, followed by an example to illustrate their usages.

  The **Hash** operator takes one operand of a tree type, and returns and integer representing the order of the operand among its siblings. If the operand has no parent, the return value is -1.

  The **square bracket operator** is used to get access to children of a tree. It takes two operands, the first on the left of the brackets and the second in the brackets. The first operand is of a tree type, and the second is either an integer index or children alias string index defined at the tree type definition. The return value is the reference to a the required child. Note that the operand inside the square bracket must be less than the degree of the tree. The behavior of an index exceeds the degree of the tree is unknown.

The **Dot** operator is used to access the data fields associated with the node. It is a binary and left-associative operator. The left operand is of a tree type, and the right operand is the name of a data fields defined at the type definition. The return value is the value of that particular field.

The **Ampersand** is used to get the degree of tree. It is a unary operator and appears in front of its operand, which is of a tree type. It returns an integer indicating the degree of the tree.

The **Caret** operator is used to get the parent reference of the operand. It is a unary operator and appears on the left side of the operand, which is of a tree type. It returns the reference of the operand's parent.

```
MyTree_t ta, tb, tc;            //MyTree_t is defined in section 3.2.2
alloc(ta, tb, tc);
ta ->(tb : tc);
int ta_order = #ta;             //ta_order = -1
int tb_order = #tb;             //tb_order = 0
int tc_order = #ta[1];          //tc_order = 1
MyTree_t tb_copy = ta[0];       //tb_copy has the same ref with tb
MyTree_t tc_copy = ta[1];       //tc_copy has the same ref with tc
MyTree_t t_err = ta[2];         //this usage could cause unknown errors
int tmp1 = ta.vint;             //tmp1 has the value of ta.vint, which is 0
float tmp2 = tb_copy.vflt;      //tmp2 has the value of tb.vflt, which is 1.0
string tmp3 = tc.vstr;          //tmp3 is "hi "
int degree = &ta;               //degree is assigned with 2
MyTree_t ta_copy = ^tb;         //ta_copy has the same ref with ta
```

3.3.7 Precedence Rules

To eliminate the possibility of ambiguity, the precedence of operators in TML are defined in Table 3.

3.4 Statements

3.4.1 Variable Declarations and Initialization Statements

Variable Declarations and Initializations are considered as statements in TML. It has the following syntax (The square brackets means optional):

type-specifier initializer-list;
initializer-list $\rightarrow$ *initializer* | *initializer-list* , *initializer*

Type Specifiers Type Specifiers can be any basic type or user-defined tree type.

For basic types, it can be:

- int - Intergers

1	()
2	. , []
2	# &
3	@ $ ^
4	* / %
5	+ -
6	< > <= >= != ==
7	!
8	&&
9	||
10	=
11	->

Table 3: Precedence Rules

- float - Floating point numbers
- char - Characters
- string - Strings
- bool - Booleans

For user-defined tree type, just write the type name identifier.

3.4.2 Initializers

An initializer contains two parts: the name of the variable and the initial value for it. The first part is only an identifier. The second part is optional (We use square brackets to represent optional), and contains an equal sign and an expression that will be evaluated and assigned to that variable.

Initializer $\rightarrow$ identifier [= *expression*]

3.4.3 Expression Statements

The expression statement is the most common one in TML. It consists of an expression and a semicolon at the end. Any expression can be used here. TML will evaluate the expression and ignore the final evaluation result.

expression;

3.4.4 Block Statements

The block statement is a list of statements surrounded by curly braces.

{ [*statement-list*] }

The *statement-list* consists of sequential statements one after another

statement-list $\rightarrow$ *statement* | *statement statement-list*

3.4.5 Conditional Statements

The conditional statements contain only the if statement with the following syntax:

```
if (expression) statement [else statement]
```

If compound statements are used for both statements, as is used mostly, the if statement can be written as follows:

```
if (x < 0)
{
    real = false;
    y = sqrt(-x);
}
else
{
    real = true;
    y = sqrt(x);
}
```

3.4.6 Iterative Statements

There are four kinds of iterative statements: while statement, do statement, for statement and foreach statement.

- while statement The while statement contains a condition-expression and a loop body. The codes in the loop body will be executed again and again as long as the evaluation result of the condition-expression is true. The condition-expression will be evaluated before each time the loop body is executed. This statement has the following syntax:

  ```
  while (expression) statement
  ```

- do statement Similar to the while statement, the do statement also contains a condition-expression and a loop body. The codes in the loop body will be executed again and again as long as the evaluation result of the condition-expression is true. But the condition-expression will be evaluated after each time the loop body is executed. This statement has the following syntax:

  ```
  do
     statement
  while (expression);
  ```

- for statement The for statement takes three expressions: init-expression, cond-expression and loop-expression. At the beginning, the init-expression will be evaluated. And then, the cond-expression is evaluated repeatedly until the result is false. For each time the evaluation has the result true, the codes of the loop body will be executed and the loop-expression will be evaluated afterward. It has the following syntax:

```
for (init-expression; cond-expression; loop-expression)
    statement
```

- foreach statement The foreach statement is used to enumerate the elements contained in an object, like all the characters in a string, all subtrees in a tree, and so on. It is mainly operated on trees. The syntax of this statement is:

  ```
  foreach variable1 in variable2 by traverse-order
      statement
  traverse-order → preorder | inorder | postorder | levelorder
  ```

  This statement will enumerate all the elements contained in variable2, and for each of the elements, it will store the element into variable1 and execute the loop body once. The order of the elements in each iteration will be determined by the *traverse-order*. And there are four kinds of tree traverse order: pre-order, in-order, post-order and level-order.

3.4.7 Other Statements

There are other statements that we may use in special conditions.

- break statements This statement is simply written as:

  ```
  break;
  ```

  It is used to immediately terminate a loop and execute the codes following the loop.

- continue statements This statement is simply written as:

  ```
  continue;
  ```

  It is used to immediately enter the next iteration of the loop ignoring the left of the codes in this iteration.

- return statements This statement is simply written as:

  ```
  return [expression];
  ```

  It is used to immediately exit a function and take expression as the return value of the function. The expression is optional as some functions do not have a return value.

- Empty statements This statement is simply written as:

  ```
  ;
  ```

  It does nothing.

3.5 Functions

3.5.1 Function Definition

TML allows only global functions and the functions are order-sensitive. It means that all functions should be defined outside all other function bodies, directly in the file scope. And if you want to call a function, that function should be defined before calling. It is allowed to define recursive functions, which will call the function itself inside the function body, but it is not allowed to define two functions that will call each other.

The function definition has the regular syntax as follows:

```
type-specifier identifier ( parameter-list )
        block-statement
```

The type-specifier is used to declare the type of the return value of the function, and it is the same as described in section 3.4.1. The identifier is the name of the function, which is used by function calls. The statement is the function body and is usually a compound statement. What's in the parentheses is an optional parameter list with the following format and each parameter should be specified type by type-specifier and name by identifier:

```
parameter-list → parameter-declaration
                 parameter-declaration , parameter-list
parameter-declaration → type-specifier identifier
```

An example of function:

```
int gcd(int a, int b)
{
        if (b == 0 || (a = a % b) == 0)
        {
                return b;
        }
        return gcd(b, a);
}
```

3.5.2 Main Function

In TML, there is an entry function where the program starts. There must be one main function in a TML program and should be defined like this:

```
void main()
{
        statement-list
}
```

3.5.3 Built-in Functions

There are mainly two built-in functions in TML, doing some regular jobs. The function names are reserved and can't be redefined.

- print function This function is used to print some information onto the screen. As a built-in function, the parameters passed to this function can be various. The definition of this function can be regarded as:

  void print(*item-list*) { ... }

  The *item-list* can be a list of items separated by commas. The items can be literals and variables of any basic type. The print function will print them on screen from left to right and move to a new line after printing all the items in the list.

- alloc function This function is used to allocate space for variables of any type of tree. Tree variables can only be used as iterators and cannot be assigned node values or children before we allocate memory space of certain size for them. As a built-in function, the parameters passed to this function can be various. The definition of this function can be regarded as:

  int alloc(*tree-list*) { ... }

The *tree-list* can be a list of trees separated by commas. The trees can be of any tree type. The alloc function will recognize the size needed for each tree and allocate the space for them from left to right and return if the allocation succeeds.

3.6 Scope

In TML, there are two finds of scopes. Codes directly written in the file has a file scope(global), starting from the current position to the end of the file. Also, each compound statement will generate a local scope confined by the two curly brackets. Identifiers can only be used within its scope.

3.7 Language Restrictions

TML has some language restrictions, which are already mentioned in previous sections. In this section, the restrictions are summarized as below.

- TML source codes consist of only **ASCII** character set.
- Sizes and value ranges for primitive types in TML is in Table 2.
- Each tree type in TML can have 99 member variables at most.
- The maximum degree of each tree type in TML is 99.

3.8 Language Formalization

TML language formalization is done in ocamlyacc format. The production rules with operator precedence rules are listed below, followed by the ocamlyacc output.

```
/* parser.mly */
/* @authors: Shuai Sun, Yan Zou, Jiabin Hu, Akash */
%{ open Type %}
%{ open Ast %}

%token <string> ID
%token IF ELSE WHILE DO FOR BREAK CONTINUE FOREACH IN BY RETURN
%token PREORDER INORDER POSTORDER LEVELORDER
%token INT_T FLOAT_T CHAR_T STRING_T BOOL_T VOID TREETYPE
%token <int>INT
%token <float>FLOAT
```

```
%token <bool>BOOL
%token <string>STRING
%token <char>CHAR
%token NULL
%token LBRACE RBRACE SEMI COLON COMMA
%token ASSIGN
%token CONNECT
/* These operators are removed
%token PLUS_ASN MINUS_ASN TIMES_ASN DIVIDE_ASN MOD_ASN
*/
%token OR AND NOT
%token NEQ GT LT LEQ GEQ EQ
%token PLUS MINUS TIMES DIVIDE MOD
%token DOLLAR AT LBRACK RBRACK DEG_AND DOT HASH FATHER
%token LPAREN RPAREN
%token EOF

%nonassoc NOELSE
%nonassoc ELSE

%right CONNECT
%right ASSIGN
/* These operators are removed
%right PLUS_ASN MINUS_ASN
%right TIMES_ASN DEVIDE_ASN MOD_ASN
*/
%left OR
%left AND
%left EQ NEQ
%left LT GT LEQ GEQ
%left PLUS MINUS   /* for binop they are left, but for unop should they be right? */
%right SIGN
%left TIMES DIVIDE MOD
%right NOT
%right AT, DOLLAR, FATHER
%right HASH, DEG_AND
%nonassoc LBRACK
%left DOT

%start program
%type <Ast.program> program      /* this type should be AST.program. just put int because AST is not fini
%%


program:
    /* nothing */                             { [] }   /* this part should be differnt
                                                          and it should be similar to MicroC unsure about it
    | program type_def                        { Treedef($2)::$1 }
    | program decl                            { Globalvar($2)::$1 }
    | program func_def                        { Funcdef($2)::$1 }

type_def:
    TREETYPE LT INT GT ID LBRACE decl_list RBRACE
                                                            { { typename = $5;
                                                                members = List.rev $7;
                                                                degree = $3;
                                                                aliases = [] }}
    | TREETYPE LT INT COMMA LBRACK alias_list RBRACK GT ID LBRACE decl_list RBRACE
                                                            { { typename = $9;
                                                                members = List.rev $11;
                                                                degree = $3;
                                                                aliases = List.rev $6 } }

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

decl_list:
    decl                                     { [$1] }
    | decl_list decl                         { $2::$1 }
```

```
decl:
    type_specifier init_list SEMI            { ($1, List.rev $2) }

type_specifier:
    INT_T                                   { Int }
    | FLOAT_T                               { Float }
    | CHAR_T                                { Char }
    | STRING_T                              { String }
    | BOOL_T                                { Boolean }
    | ID                                    { Tree_type($1) }
                | VOID                                  { Void }

/*
return_type:
    type_specifier                          { $1 }
    | VOID                                  { Void }
*/

init_list:
    init                                    { [$1] }
    | init_list COMMA init                  { $3::$1 }

init:
    ID                                      { WithoutInit($1) }
    | ID ASSIGN expr                        { WithInit($1 ,$3) }

func_def:
    type_specifier ID LPAREN para_list RPAREN stmt_block           { {  return_type = $1;
                                                                          fname = $2;
                                                                          params = List.rev $4;



para_list:
    /* nothing */                                              {[]}
    | para_decl                             { [$1] }
    | para_list COMMA para_decl             { $3::$1 }

para_decl:
    type_specifier ID                       { ($1, $2) }

stmt_block:
    LBRACE stmt_list RBRACE                 { List.rev $2 }

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

stmt:
    expr SEMI                                                       { Expr($1) }
    | decl                                                  { Vardecl($1) }
    | stmt_block                                            { Block($1) }
    | IF LPAREN expr RPAREN stmt %prec NOELSE               { If($3, $5, Empty) }
    | IF LPAREN expr RPAREN stmt ELSE stmt                  { If($3, $5, $7) }
    | WHILE LPAREN expr RPAREN stmt                         { While($3, $5) }
    | DO stmt WHILE LPAREN expr RPAREN SEMI                 { Do($2, $5) }
    | FOR LPAREN expr SEMI expr SEMI expr RPAREN stmt       { For($3, $5, $7, $9) }
    /*| FOREACH LPAREN ID IN expr BY trvs_order RPAREN stmt     { Foreach($3, $5, $7, $9) } */
    | FOREACH ID IN expr BY trvs_order stmt                 { Foreach($2, $4, $6, $7) }
                                                                  /* drop the parenthesis */
    | BREAK SEMI                                            { Break }
    | CONTINUE SEMI                                         { Continue }
    | RETURN expr SEMI                                      { Return($2) }
    | RETURN SEMI                                           { ReturnVoid }
    | SEMI                                                  { Empty }      /*No action really.. ?*/

trvs_order:
    INORDER                                             { Inorder }
                                                          /*I doubt what exactly these should contain..
    | PREORDER                                                { Preorder }
    | POSTORDER                                               { Postorder }
    | LEVELORDER                                              { Levelorder }

expr:
    | literal                         { Literal($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 MOD expr                   { Binop($1, Mod, $3) }

    | expr GT expr                    { Binop($1, Greater_than, $3) }
    | expr LT expr                    { Binop($1, Less_than, $3) }
    | expr GEQ expr                   { Binop($1, Geq, $3) }
    | expr LEQ expr                   { Binop($1, Leq, $3) }
    | expr NEQ expr                   { Binop($1, Neq, $3) }
    | expr EQ expr                    { Binop($1, Equal, $3) }

    | expr AND expr                   { Binop($1, And, $3) }
    | expr OR expr                    { Binop($1, Or, $3) }

    | PLUS expr                       { Uniop(Add, $2) }
    | MINUS expr                      { Uniop(Sub, $2 ) }
    | AT expr                                 { Uniop(At, $2) }
    | DOLLAR expr                     { Uniop(Dollar, $2) }
    | FATHER expr                     { Uniop(Father, $2) }
    | NOT expr                        { Uniop(Not, $2) }
    | HASH expr                       { Uniop(Hsh, $2) }
    | DEG_AND expr                    { Uniop(Deg_a, $2) }

    | lvalue                          { $1 }

    | lvalue ASSIGN expr              { Assign($1, $3) }

    | lvalue CONNECT LPAREN node_list RPAREN  { Conn($1, List.rev  $4) }

    | LPAREN expr RPAREN              { $2 }
    | ID LPAREN arg_list RPAREN       { Call($1, List.rev $3) }


literal:
    INT                               { IntLit($1) }
    | FLOAT                           { FloatLit($1) }
    | STRING                          { StringLit($1) }
    | CHAR                            { CharLit($1) }
    | BOOL                            { BoolLit($1) }
    | NULL                            { TreeLit }
                                      /*We dont have a NULL type defined in ast yet*/

node_list:
    expr                              { [$1] }
    | node_list COLON expr            { $3::$1 }

lvalue:
    ID                                { Id($1) }
    | expr DOT ID                     { Binop($1, Dot, Id($3)) }
    | expr LBRACK expr RBRACK         { Binop($1, Child, $3) }

arg_list:
    /* nothing */                     { [] }
    | expr                            { [$1] }
    | arg_list COMMA expr             { $3::$1 }
```

```
 And the output of the parser in file "parser.output" is:
   72 terminals, 22 nonterminals
   91 grammar rules, 190 states
```

4 Project Plan

4.1 Developing Process

Our team met once a week on Wednesday after class in regular weeks. In weeks that had more issues to discuss or tasks than normal, we arranged more meeting in the weekend. On the meetings, we first went through things we accomplished the week before, and then set milestones and time plans to the next week. When finalizing issues, we used to take notes and forward to every member of the team.

To communicate among members, we used Google Group with group emails available to everyone. To maintain a version control over our source code, our team used the Google Code repository as put our TML project open-source. Most of the code of the project was written separately. We used vim, Eclipse as source text editors while developing. Team members committed all their work to the Google Code repository upon accomplishment.

TML compiler was written in OCaml, while the interpreter was in Java. As a result, our TML is a platform independent language. The byte code generated from compiler is now in text format.

The authors are in the comment at the beginning of each file, with a few exceptions. In our development directory, we provided test cases for each module along with its expected outputs, which could be verified by running the test suit scripts.

4.2 Style Guide

The general style guide was given below. Note that because we are using the OcaIDE on Eclipse developing environment, which provides automated indent feature, and we used the automated indent feature. So there is not much issues on the programming style.

4.2.1 O'Caml source

- Variables are named in lowercase with spaces replaced by underscores.
- Types are named using lowercase letters with spaces replaced by underscores.
- Tokens are named using uppercase letters without spaces.
- Constructors are named as first letters of words in uppercase and others in lowercase.
- Modules are named as first letters of words in uppercase and others in lowercase.
- Each tab takes the size of four spaces, and not converting to spaces.
- Comments about detailed implementation or questions are at the place where it occurs.
- Normally, use fewer than 125 characters in a single line.
- Keep "let" and "in" aligned if the function takes multiple lines.
- Keep functions aligned if they are in the same level of scope.

4.2.2 Java

Strictly use the automated code style in Eclipse.

4.2.3 TML source

- Write the global variables and tree definitions in the beginning of the file.
- Write the functions before it is first called. (required by the compiler)
- Write the code in C-like style, if possible.

4.2.4 Directory Arrangement

- Source files of TML compiler should be under ./src/TML_Compiler/ directory.
- Source files of TML interpreter should be under ./src/TML_Interpreter/ directory.
- Test case TML source files should be under ./test/SrcCode/ directory.
- All documentation of the TML project should be under ./doc/ directory, each in a separate folder.
- The Makefile of the TML project is under ./src/ directory.

4.3 Project Time line

Sept.28	Decided on project planning and finished proposal.
Oct. 31	Finished first version of scanner, parser and definition of ast.
	Finalized Language Reference Manual.
Nov. 15	Added test cases.
Dec. 6	Finished a simple version compiler without supporting tree features.
Dec. 15	Finished type checking and code generator of non-tree part;
	Finished a simple version interpreter.
Dec. 20	Finished coding of the project.
Dec. 21	Finalized the final report.

4.4 Member Roles and Responsibilities

4.4.1 Jiabin Hu

Participated in all discussions and contributed to the project management. Made great suggestions on design issues. Made reasonable arguments in language design. Wrote test cases for the expression features in TML. Designed the interpreter with the group and implemented the interpreter in Java. Helped and made suggestions in the design of the TML byte code. Fixed many bugs and rearranged the project repository directory. Modified the Makefiles.

Wrote a version of the Proposal. Wrote the expression part in the LRM.

4.4.2 Akash Sharma

Participated in all discussions. Have a great understanding of the design issues and made suggestions on language designing. Wrote test cases for primitive type operations in TML. Wrote the definition of the AST. Helped figuring out the conflicts in parser and AST. Tested the analyzer of the compiler. Wrote the final complex test cases for TML.

Wrote a version of the Proposal. Wrote the primitive type part, and language restriction part in the LRM. Made the presentation slice.

4.4.3 Shuai Sun

Managed the group meetings and was responsible for the project management. Initiated the SVN repository. Initiated the idea of TML and making all things as tree. Made contributions to language design. Wrote the scanner and parser. Working with Yan, designed the compiler, including starting with the parts similar with MICROC, designing the byte codes, translating to the SAST, type checking and code generating. Wrote the test cases for tree type features in TML. Fixed bugs and reorganized source codes.

Wrote the final version of the Proposal. Wrote the type part in the LRM, and constructed the LRM portions into one. Wrote this final report.

4.4.4 Yan Zou

Participated in all discussions. Made great contribution on language design. Wrote the scanner test program. Designed the compiler, including analyzer, byte code and its generator. Made the mainly contribution to the implementation of compiler, including analyzer, byte code, and definition of SAST, all starting from scratch. Wrote the analyzer test program. Wrote the test cases for control flow of TML. Fixed bugs in the compiler. Wrote and modified the Makefiles.

Worked on the final version of the Proposal. Wrote the statement part in LRM.

4.5 Developing Tools and Environment

The team used O'Caml, Java, and bash scripting to complete the whole project. Our developing environment included Mac OS X, Windows Vista and Windows 7. We used OcalDE on Eclipse as our O'Caml developing tool, and Eclipse Java Development Tools as our Java developing tool. Besides, we used Google Code as our SVN server, and Subclipse on Eclipse and Tortoise as SVN client software. Our team used the Google Group to hold all our discussions and notifications. Our team also used Google Docs to share discussion notes and design drafts. The documentation of the TML project used TeXworks on both Mac OS and Windows.

4.6 Project Log

```
------------------------------------------------------------------------
r155 | marksun1988@gmail.com | 2011-12-22 18:05:15 -0500 (Thu, 22 Dec 2011) | 4 lines

1. Modified final report.
2. Modified LRM: let it print the current parser.mly
3. Modified test.bash: add clean operation
4. Modified and added README/readme.txt
------------------------------------------------------------------------
r153 | probas.322@gmail.com | 2011-12-21 14:15:31 -0500 (Wed, 21 Dec 2011) | 3 lines

small fix of analyzer
add analyzer_test to test/README
modify bst.tml to make it a little more concise
------------------------------------------------------------------------
r152 | hu.jiabin88@gmail.com | 2011-12-21 11:37:27 -0500 (Wed, 21 Dec 2011) | 2 lines

Update the readme in test folder

------------------------------------------------------------------------
r151 | hu.jiabin88@gmail.com | 2011-12-21 11:28:40 -0500 (Wed, 21 Dec 2011) | 3 lines

Update tml scripts to call tml.jar in the same directory as the script.
Change bst, fix 'bool' from 'boolean'
```

```
------------------------------------------------------------------------
r150 | hu.jiabin88@gmail.com | 2011-12-21 10:50:28 -0500 (Wed, 21 Dec 2011) | 2 lines

Add a bst sample/test case

------------------------------------------------------------------------
r149 | hu.jiabin88@gmail.com | 2011-12-21 09:37:17 -0500 (Wed, 21 Dec 2011) | 2 lines

Add Interpreter scripts, readme file to src folder.

------------------------------------------------------------------------
r148 | hu.jiabin88@gmail.com | 2011-12-21 09:10:47 -0500 (Wed, 21 Dec 2011) | 2 lines

Modifiy makefiles to output binaries to src folder

------------------------------------------------------------------------
r147 | hu.jiabin88@gmail.com | 2011-12-21 08:48:11 -0500 (Wed, 21 Dec 2011) | 2 lines

Check in some test cases.. Forgot to do that yesterday.

------------------------------------------------------------------------
r146 | marksun1988@gmail.com | 2011-12-21 06:56:20 -0500 (Wed, 21 Dec 2011) | 2 lines

1. modified parser to remove the parenthesis in foreach statement
2. first full version of the final report, along with figures and codes needed.
------------------------------------------------------------------------
r145 | marksun1988@gmail.com | 2011-12-21 05:39:33 -0500 (Wed, 21 Dec 2011) | 1 line

add test cases for foreach loop
------------------------------------------------------------------------
r144 | akash1531@gmail.com | 2011-12-21 01:52:33 -0500 (Wed, 21 Dec 2011) | 1 line

final
------------------------------------------------------------------------
r143 | akash1531@gmail.com | 2011-12-21 01:51:54 -0500 (Wed, 21 Dec 2011) | 1 line

final without errors
------------------------------------------------------------------------
r142 | hu.jiabin88@gmail.com | 2011-12-21 01:12:23 -0500 (Wed, 21 Dec 2011) | 1 line

Delete non-source files.
------------------------------------------------------------------------
r141 | akash1531@gmail.com | 2011-12-21 01:03:39 -0500 (Wed, 21 Dec 2011) | 1 line

out
------------------------------------------------------------------------
r140 | akash1531@gmail.com | 2011-12-21 01:03:20 -0500 (Wed, 21 Dec 2011) | 1 line

check
------------------------------------------------------------------------
r139 | akash1531@gmail.com | 2011-12-21 00:53:59 -0500 (Wed, 21 Dec 2011) | 1 line

bug
------------------------------------------------------------------------
r138 | probas.322@gmail.com | 2011-12-21 00:39:02 -0500 (Wed, 21 Dec 2011) | 4 lines

break/continue bug fixed!!!!
It can pop correct number of variables before leaving loop scope
move the job of counting variable from generator to analyzer
sast changed to record number of local variables before break/continue in the loop
------------------------------------------------------------------------
r137 | akash1531@gmail.com | 2011-12-21 00:27:12 -0500 (Wed, 21 Dec 2011) | 1 line

test
------------------------------------------------------------------------
r136 | akash1531@gmail.com | 2011-12-21 00:22:00 -0500 (Wed, 21 Dec 2011) | 1 line

continue.output
------------------------------------------------------------------------
r135 | akash1531@gmail.com | 2011-12-21 00:21:43 -0500 (Wed, 21 Dec 2011) | 1 line
```

```
continue.tml
------------------------------------------------------------------------
r134 | akash1531@gmail.com | 2011-12-20 23:54:10 -0500 (Tue, 20 Dec 2011) | 1 line

corrected
------------------------------------------------------------------------
r133 | hu.jiabin88@gmail.com | 2011-12-20 23:45:09 -0500 (Tue, 20 Dec 2011) | 2 lines

Clean up test folder

------------------------------------------------------------------------
r132 | hu.jiabin88@gmail.com | 2011-12-20 23:40:26 -0500 (Tue, 20 Dec 2011) | 3 lines

Fix a bug in scanner
Fix several test cases

------------------------------------------------------------------------
r131 | hu.jiabin88@gmail.com | 2011-12-20 23:03:25 -0500 (Tue, 20 Dec 2011) | 3 lines

Implemented foreach and nested foreach.
Sign my names.
Last commit barring bugs fixes.
------------------------------------------------------------------------
r130 | akash1531@gmail.com | 2011-12-20 22:48:03 -0500 (Tue, 20 Dec 2011) | 1 line


------------------------------------------------------------------------
r129 | akash1531@gmail.com | 2011-12-20 22:47:48 -0500 (Tue, 20 Dec 2011) | 1 line

smaller loop
------------------------------------------------------------------------
r128 | akash1531@gmail.com | 2011-12-20 22:44:41 -0500 (Tue, 20 Dec 2011) | 1 line

added output
------------------------------------------------------------------------
r127 | akash1531@gmail.com | 2011-12-20 22:44:23 -0500 (Tue, 20 Dec 2011) | 1 line

added output
------------------------------------------------------------------------
r126 | hu.jiabin88@gmail.com | 2011-12-20 22:25:23 -0500 (Tue, 20 Dec 2011) | 1 line

Add tree iterators.
------------------------------------------------------------------------
r125 | marksun1988@gmail.com | 2011-12-20 22:11:59 -0500 (Tue, 20 Dec 2011) | 1 line

add some intermediate tex file to ignore list
------------------------------------------------------------------------
r124 | marksun1988@gmail.com | 2011-12-20 22:09:01 -0500 (Tue, 20 Dec 2011) | 3 lines

first commit the final report: not finished yet.

add the proposal folder to version control.
------------------------------------------------------------------------
r123 | probas.322@gmail.com | 2011-12-20 21:48:44 -0500 (Tue, 20 Dec 2011) | 4 lines

modify bytecodes for foreach - Nxt of int
break and continue for foreach
check the tree degree for inorder foreach
check the number of aliases for tree type
------------------------------------------------------------------------
r122 | hu.jiabin88@gmail.com | 2011-12-20 19:42:46 -0500 (Tue, 20 Dec 2011) | 1 line

Implemented everything except foreach.
------------------------------------------------------------------------
r121 | probas.322@gmail.com | 2011-12-20 16:52:36 -0500 (Tue, 20 Dec 2011) | 1 line

analyzer: reject all variables and parameters of type void
------------------------------------------------------------------------
r120 | probas.322@gmail.com | 2011-12-20 16:35:26 -0500 (Tue, 20 Dec 2011) | 2 lines
```

```
allocation return bug fixed
add code to analyzer to check if function parameters have the same name
------------------------------------------------------------------------
r119 | hu.jiabin88@gmail.com | 2011-12-20 16:23:38 -0500 (Tue, 20 Dec 2011) | 1 line

Implement tree instructions.
------------------------------------------------------------------------
r118 | probas.322@gmail.com | 2011-12-20 16:09:43 -0500 (Tue, 20 Dec 2011) | 2 lines

another bug of alloc found by Jiabin - the newly allocated tree bad been simply discarded
The bug is fixed by changing it into assignments
------------------------------------------------------------------------
r117 | probas.322@gmail.com | 2011-12-20 11:54:40 -0500 (Tue, 20 Dec 2011) | 1 line

small bug found by Jiabin in alloc function
------------------------------------------------------------------------
r116 | hu.jiabin88@gmail.com | 2011-12-20 11:52:46 -0500 (Tue, 20 Dec 2011) | 2 lines

Add two test cases

------------------------------------------------------------------------
r115 | probas.322@gmail.com | 2011-12-20 00:05:52 -0500 (Tue, 20 Dec 2011) | 6 lines

analyzer:
1. deal with foreach
2. add block scope to all if, for, while and do statement body
3. check redeclaration (function, tree type, parameter, same scope local, global)
generator:
can generate codes for tree connection and foreach(Jsr -2) now
------------------------------------------------------------------------
r114 | probas.322@gmail.com | 2011-12-18 01:17:32 -0500 (Sun, 18 Dec 2011) | 1 line

The generator can now generate bytecodes to manipulate trees
------------------------------------------------------------------------
r113 | akash1531@gmail.com | 2011-12-18 00:22:54 -0500 (Sun, 18 Dec 2011) | 1 line

complex tree declaration
------------------------------------------------------------------------
r112 | akash1531@gmail.com | 2011-12-18 00:16:13 -0500 (Sun, 18 Dec 2011) | 1 line

another test case
------------------------------------------------------------------------
r111 | akash1531@gmail.com | 2011-12-17 22:53:56 -0500 (Sat, 17 Dec 2011) | 1 line

break inside for loop
------------------------------------------------------------------------
r110 | akash1531@gmail.com | 2011-12-17 22:48:55 -0500 (Sat, 17 Dec 2011) | 1 line

analyzer test directory
------------------------------------------------------------------------
r109 | akash1531@gmail.com | 2011-12-17 22:48:22 -0500 (Sat, 17 Dec 2011) | 1 line

output directory
------------------------------------------------------------------------
r108 | akash1531@gmail.com | 2011-12-17 22:04:40 -0500 (Sat, 17 Dec 2011) | 1 line

combination of functions and for loops
------------------------------------------------------------------------
r107 | akash1531@gmail.com | 2011-12-17 20:07:19 -0500 (Sat, 17 Dec 2011) | 1 line

comment
------------------------------------------------------------------------
r106 | akash1531@gmail.com | 2011-12-17 19:38:53 -0500 (Sat, 17 Dec 2011) | 1 line

changed comments
------------------------------------------------------------------------
r105 | akash1531@gmail.com | 2011-12-17 19:33:31 -0500 (Sat, 17 Dec 2011) | 1 line

changed comments
```

```
------------------------------------------------------------------------
r104 | hu.jiabin88@gmail.com | 2011-12-17 16:57:25 -0500 (Sat, 17 Dec 2011) | 1 line

Update standard output for several test cases
------------------------------------------------------------------------
r103 | hu.jiabin88@gmail.com | 2011-12-17 16:52:52 -0500 (Sat, 17 Dec 2011) | 2 lines

Implemented Pss.
Modified scanner to recognize escape sequence in string literals.
------------------------------------------------------------------------
r102 | hu.jiabin88@gmail.com | 2011-12-17 11:48:04 -0500 (Sat, 17 Dec 2011) | 3 lines

Update scanner to accept empty string.
Update a test case.

------------------------------------------------------------------------
r101 | hu.jiabin88@gmail.com | 2011-12-17 11:40:15 -0500 (Sat, 17 Dec 2011) | 1 line

Implement psf, psc, pss, TMLTree
------------------------------------------------------------------------
r100 | probas.322@gmail.com | 2011-12-17 02:39:16 -0500 (Sat, 17 Dec 2011) | 4 lines

tree type checking of analyzer
output ascii of char/string
\' bug in scanner
wrong test cases
------------------------------------------------------------------------
r99 | hu.jiabin88@gmail.com | 2011-12-16 11:33:12 -0500 (Fri, 16 Dec 2011) | 1 line

Updating standard output for several test cases to eliminate extra new line at the end of file.
------------------------------------------------------------------------
r98 | hu.jiabin88@gmail.com | 2011-12-16 11:05:30 -0500 (Fri, 16 Dec 2011) | 2 lines

Update the test script to compare results. Add option to run single tests.

------------------------------------------------------------------------
r97 | hu.jiabin88@gmail.com | 2011-12-16 10:41:12 -0500 (Fri, 16 Dec 2011) | 1 line


------------------------------------------------------------------------
r96 | probas.322@gmail.com | 2011-12-15 22:45:41 -0500 (Thu, 15 Dec 2011) | 1 line

modify the original test.bash and add analyzer_test.bash
------------------------------------------------------------------------
r95 | probas.322@gmail.com | 2011-12-15 22:41:32 -0500 (Thu, 15 Dec 2011) | 5 lines

append return to the end of all functions(default value)
return void will return 0
fix the recursion error
unary operation on + and -
bug of the father operator of the parser
------------------------------------------------------------------------
r94 | probas.322@gmail.com | 2011-12-15 00:07:35 -0500 (Thu, 15 Dec 2011) | 4 lines

The first complete version of generator.
Bytecode has been changed!!!
The top level can now take in one argument (input file name) and generate the output file name according
Some revision of the analyzer.
------------------------------------------------------------------------
r93 | probas.322@gmail.com | 2011-12-14 17:45:51 -0500 (Wed, 14 Dec 2011) | 3 lines

The first complete version of analyzer.ml and analyzer_test.ml
Add analyzer_test to Makefile
analyzer_test can print out the SAST
------------------------------------------------------------------------
r92 | hu.jiabin88@gmail.com | 2011-12-14 17:30:13 -0500 (Wed, 14 Dec 2011) | 1 line

Remove dead code in parser.mly
------------------------------------------------------------------------
r91 | hu.jiabin88@gmail.com | 2011-12-14 16:47:01 -0500 (Wed, 14 Dec 2011) | 2 lines
```

```
Update make clean to clean analyzer_test

------------------------------------------------------------------------
r90 | hu.jiabin88@gmail.com | 2011-12-14 16:32:50 -0500 (Wed, 14 Dec 2011) | 1 line

Update interpreter
------------------------------------------------------------------------
r89 | probas.322@gmail.com | 2011-12-13 18:35:31 -0500 (Tue, 13 Dec 2011) | 2 lines

Adjust the code structures.
Create 5 new files and do small modifications to all the existing compiler files
------------------------------------------------------------------------
r88 | marksun1988@gmail.com | 2011-12-07 18:20:18 -0500 (Wed, 07 Dec 2011) | 1 line

add test.byte
------------------------------------------------------------------------
r87 | hu.jiabin88@gmail.com | 2011-12-07 18:15:29 -0500 (Wed, 07 Dec 2011) | 1 line

Add instruction Glb.
------------------------------------------------------------------------
r86 | hu.jiabin88@gmail.com | 2011-12-07 18:08:17 -0500 (Wed, 07 Dec 2011) | 1 line

Change the names of some instruction, change the meaning of Ent.
------------------------------------------------------------------------
r85 | probas.322@gmail.com | 2011-12-06 22:58:39 -0500 (Tue, 06 Dec 2011) | 16 lines

A complete version of compiler along with other changes

Makefile:
        Add dependence of compiler and bytecode
ast.mli:
        Move Assign from "type op" to "type expr"
bytecode.mli:
        Add Glb to indicate the number of global variables
parser.mly:
        if statement without else: "Block[]" -> "Empty"
        value ASSIGN expr: "Binop($1, Equal, $3)" -> "Assign($1, $3)"
compiler.ml:
        Add compilation of loading/storing local/global variables,
        Break/Continue statement, function calls, entry function,
        main function, built-in functions.
        All the function of MicroC has been implemented.
------------------------------------------------------------------------
r84 | probas.322@gmail.com | 2011-12-05 00:20:03 -0500 (Mon, 05 Dec 2011) | 1 line

bytecode.ml, tml.ml(top level), first simple version of compiler, and modifier parser(Reverse the order o
------------------------------------------------------------------------
r83 | hu.jiabin88@gmail.com | 2011-12-02 23:03:30 -0500 (Fri, 02 Dec 2011) | 1 line

Modify test scripts to include interpreter tests
------------------------------------------------------------------------
r82 | hu.jiabin88@gmail.com | 2011-12-02 23:02:59 -0500 (Fri, 02 Dec 2011) | 1 line

Fix a bug in interpreter Makefile
------------------------------------------------------------------------
r81 | hu.jiabin88@gmail.com | 2011-12-02 22:34:36 -0500 (Fri, 02 Dec 2011) | 1 line

Missed a file in last commit..oops.
------------------------------------------------------------------------
r80 | hu.jiabin88@gmail.com | 2011-12-02 22:32:52 -0500 (Fri, 02 Dec 2011) | 1 line

Change directory structure, add Makefiles for different directories.
------------------------------------------------------------------------
r79 | hu.jiabin88@gmail.com | 2011-12-02 22:09:03 -0500 (Fri, 02 Dec 2011) | 1 line

Add test cases for byte code interpreter.
------------------------------------------------------------------------
r78 | hu.jiabin88@gmail.com | 2011-12-02 22:05:18 -0500 (Fri, 02 Dec 2011) | 1 line
```

```
Implemented execute(). Now we're able to execute gcd().
------------------------------------------------------------------------
r77 | hu.jiabin88@gmail.com | 2011-12-02 20:40:54 -0500 (Fri, 02 Dec 2011) | 1 line

Implement the data structures and reading from input.
------------------------------------------------------------------------
r76 | hu.jiabin88@gmail.com | 2011-12-02 17:59:00 -0500 (Fri, 02 Dec 2011) | 1 line

First commit for interpreter.
------------------------------------------------------------------------
r75 | hu.jiabin88@gmail.com | 2011-12-02 17:52:15 -0500 (Fri, 02 Dec 2011) | 1 line

Initial import.
------------------------------------------------------------------------
r74 | hu.jiabin88@gmail.com | 2011-12-02 17:50:36 -0500 (Fri, 02 Dec 2011) | 1 line

Change directory structure
------------------------------------------------------------------------
r73 | marksun1988@gmail.com | 2011-11-30 20:00:55 -0500 (Wed, 30 Nov 2011) | 2 lines

Ast works at this point.
Ast Tests needed.
------------------------------------------------------------------------
r72 | marksun1988@gmail.com | 2011-11-30 19:28:17 -0500 (Wed, 30 Nov 2011) | 1 line

not yet finished
------------------------------------------------------------------------
r71 | akash1531@gmail.com | 2011-11-30 17:32:16 -0500 (Wed, 30 Nov 2011) | 1 line

hello
------------------------------------------------------------------------
r70 | hu.jiabin88@gmail.com | 2011-11-17 14:02:41 -0500 (Thu, 17 Nov 2011) | 1 line


------------------------------------------------------------------------
r69 | akash1531@gmail.com | 2011-11-16 18:30:15 -0500 (Wed, 16 Nov 2011) | 1 line

checked in
------------------------------------------------------------------------
r68 | hu.jiabin88@gmail.com | 2011-11-16 15:59:24 -0500 (Wed, 16 Nov 2011) | 1 line

Add and modify basic operator test cases
------------------------------------------------------------------------
r67 | akash1531@gmail.com | 2011-11-16 00:38:11 -0500 (Wed, 16 Nov 2011) | 1 line

additions
------------------------------------------------------------------------
r66 | akash1531@gmail.com | 2011-11-15 23:53:30 -0500 (Tue, 15 Nov 2011) | 1 line

some changes
------------------------------------------------------------------------
r65 | marksun1988@gmail.com | 2011-11-15 23:37:20 -0500 (Tue, 15 Nov 2011) | 2 lines

modify examples:
comma to colon.
------------------------------------------------------------------------
r64 | marksun1988@gmail.com | 2011-11-15 23:34:41 -0500 (Tue, 15 Nov 2011) | 1 line

add a comment in. still leave the error open for discussion.
------------------------------------------------------------------------
r63 | akash1531@gmail.com | 2011-11-15 23:31:20 -0500 (Tue, 15 Nov 2011) | 1 line

changed but has problems
------------------------------------------------------------------------
r62 | hu.jiabin88@gmail.com | 2011-11-15 23:23:50 -0500 (Tue, 15 Nov 2011) | 4 lines

Update LRM to specify that in tree construction expression, children are seperated by COLONs, instead of
Update test cases to reflect this change.
```

```
------------------------------------------------------------------------
r61 | hu.jiabin88@gmail.com | 2011-11-15 23:04:08 -0500 (Tue, 15 Nov 2011) | 2 lines

Modify makefile and several test cases

------------------------------------------------------------------------
r60 | akash1531@gmail.com | 2011-11-15 21:39:46 -0500 (Tue, 15 Nov 2011) | 1 line

changed file
------------------------------------------------------------------------
r59 | marksun1988@gmail.com | 2011-11-15 13:54:31 -0500 (Tue, 15 Nov 2011) | 1 line

add some more tree op test cases
------------------------------------------------------------------------
r58 | akash1531@gmail.com | 2011-11-15 02:14:58 -0500 (Tue, 15 Nov 2011) | 1 line

yet another attempt
------------------------------------------------------------------------
r57 | marksun1988@gmail.com | 2011-11-15 01:09:53 -0500 (Tue, 15 Nov 2011) | 1 line

add test cases for tree type definition and some tree operators
------------------------------------------------------------------------
r56 | marksun1988@gmail.com | 2011-11-14 23:48:04 -0500 (Mon, 14 Nov 2011) | 2 lines

fixed the scanner bug for \n \b \r \t \\
add a test case to test these chars.
------------------------------------------------------------------------
r55 | marksun1988@gmail.com | 2011-11-14 22:36:21 -0500 (Mon, 14 Nov 2011) | 2 lines

remove the following operators
PLUS_ASN MINUS_ASN TIMES_ASN DIVIDE_ASN MOD_ASN
------------------------------------------------------------------------
r54 | hu.jiabin88@gmail.com | 2011-11-14 19:18:11 -0500 (Mon, 14 Nov 2011) | 2 lines

Update Makefile again, apparently I have nothing better to do.

------------------------------------------------------------------------
r53 | hu.jiabin88@gmail.com | 2011-11-14 19:03:42 -0500 (Mon, 14 Nov 2011) | 2 lines

Make makefile more intelligent.

------------------------------------------------------------------------
r52 | hu.jiabin88@gmail.com | 2011-11-14 18:25:45 -0500 (Mon, 14 Nov 2011) | 2 lines

Fix scanner bug that extracts incorrect part of matched string for character and string literal.

------------------------------------------------------------------------
r51 | hu.jiabin88@gmail.com | 2011-11-14 18:19:27 -0500 (Mon, 14 Nov 2011) | 2 lines

Fix parser bug-decl in stmt derivation should not be followed by a semicolon.

------------------------------------------------------------------------
r50 | probas.322@gmail.com | 2011-11-14 00:54:43 -0500 (Mon, 14 Nov 2011) | 4 lines

1. Simple compiler
2. Simple Makefile for compiler and scanner_test
3. Test cases for statements and functions
4. Test scripts for compiler and scanner(to debug) and README
------------------------------------------------------------------------
r49 | hu.jiabin88@gmail.com | 2011-11-13 00:17:36 -0500 (Sun, 13 Nov 2011) | 1 line

Add test cases for arithmetic_op and logical_op
------------------------------------------------------------------------
r48 | akash1531@gmail.com | 2011-11-08 02:33:19 -0500 (Tue, 08 Nov 2011) | 1 line

added again
------------------------------------------------------------------------
r47 | akash1531@gmail.com | 2011-11-08 02:12:36 -0500 (Tue, 08 Nov 2011) | 1 line

added at el.. :)
```

```
------------------------------------------------------------------------
r46 | akash1531@gmail.com | 2011-11-08 02:11:25 -0500 (Tue, 08 Nov 2011) | 1 line

based on Yan's recent comments
------------------------------------------------------------------------
r45 | akash1531@gmail.com | 2011-11-08 01:39:00 -0500 (Tue, 08 Nov 2011) | 1 line

some more additions
------------------------------------------------------------------------
r44 | akash1531@gmail.com | 2011-11-08 01:01:00 -0500 (Tue, 08 Nov 2011) | 1 line

removed multiple foreach statements
------------------------------------------------------------------------
r43 | hu.jiabin88@gmail.com | 2011-10-31 16:18:39 -0400 (Mon, 31 Oct 2011) | 1 line


------------------------------------------------------------------------
r42 | probas.322@gmail.com | 2011-10-31 14:53:45 -0400 (Mon, 31 Oct 2011) | 1 line

Final revision of LRM
------------------------------------------------------------------------
r41 | probas.322@gmail.com | 2011-10-31 12:12:10 -0400 (Mon, 31 Oct 2011) | 1 line

add parser_pdf.mly to append yacc file to LRM
------------------------------------------------------------------------
r40 | probas.322@gmail.com | 2011-10-31 12:10:37 -0400 (Mon, 31 Oct 2011) | 1 line

Add the main and built-in functions in LRM, append the yacc file to the LRM, add authors to yacc
------------------------------------------------------------------------
r39 | marksun1988@gmail.com | 2011-10-30 03:35:20 -0400 (Sun, 30 Oct 2011) | 1 line


------------------------------------------------------------------------
r38 | marksun1988@gmail.com | 2011-10-30 03:34:13 -0400 (Sun, 30 Oct 2011) | 3 lines

revised the LRM.

Add alloc in scanner.
------------------------------------------------------------------------
r37 | marksun1988@gmail.com | 2011-10-29 22:02:09 -0400 (Sat, 29 Oct 2011) | 1 line


------------------------------------------------------------------------
r36 | marksun1988@gmail.com | 2011-10-29 21:54:23 -0400 (Sat, 29 Oct 2011) | 1 line

add main and print as id tokens different from user-defined ids.
------------------------------------------------------------------------
r35 | hu.jiabin88@gmail.com | 2011-10-29 16:23:29 -0400 (Sat, 29 Oct 2011) | 2 lines

Add test cases for treetype declaration.
Fix the parser rules for treetype declaration.
------------------------------------------------------------------------
r34 | hu.jiabin88@gmail.com | 2011-10-29 16:16:37 -0400 (Sat, 29 Oct 2011) | 5 lines

parser.mly:
assign a {0} for every /* nothing */

scanner.mll:
move the ID to the end so that keywords and literals can be matched first.
------------------------------------------------------------------------
r33 | probas.322@gmail.com | 2011-10-29 16:08:43 -0400 (Sat, 29 Oct 2011) | 1 line

a test for scanner
------------------------------------------------------------------------
r32 | hu.jiabin88@gmail.com | 2011-10-29 16:00:19 -0400 (Sat, 29 Oct 2011) | 2 lines

Added two test cases.
Remove code at the end of scanner.
------------------------------------------------------------------------
r31 | hu.jiabin88@gmail.com | 2011-10-29 15:41:05 -0400 (Sat, 29 Oct 2011) | 7 lines
```

```
paresr.mly:
Added EOF as a token
Changed each rule as type of integer

scanner.mll:
Fixed match rules for string and char literals
Fixed a typo.
------------------------------------------------------------------------
r30 | hu.jiabin88@gmail.com | 2011-10-29 14:52:30 -0400 (Sat, 29 Oct 2011) | 1 line

Add sample test cases.
------------------------------------------------------------------------
r29 | probas.322@gmail.com | 2011-10-29 14:42:39 -0400 (Sat, 29 Oct 2011) | 1 line

Solve all the conflicts
------------------------------------------------------------------------
r28 | akash1531@gmail.com | 2011-10-29 12:14:04 -0400 (Sat, 29 Oct 2011) | 2 lines

Changed Foreach error as told by Yan
so now....we have expr replace (expr in expr by _order do)
------------------------------------------------------------------------
r27 | probas.322@gmail.com | 2011-10-29 12:00:40 -0400 (Sat, 29 Oct 2011) | 1 line

Fixed some small problems.
------------------------------------------------------------------------
r26 | probas.322@gmail.com | 2011-10-29 11:43:10 -0400 (Sat, 29 Oct 2011) | 1 line

Some revisions on Statements and functions
------------------------------------------------------------------------
r25 | hu.jiabin88@gmail.com | 2011-10-29 11:29:08 -0400 (Sat, 29 Oct 2011) | 1 line

Add missing operators.
------------------------------------------------------------------------
r24 | marksun1988@gmail.com | 2011-10-28 05:11:16 -0400 (Fri, 28 Oct 2011) | 1 line

add non-source-code files to ignore list
------------------------------------------------------------------------
r23 | marksun1988@gmail.com | 2011-10-28 05:09:09 -0400 (Fri, 28 Oct 2011) | 3 lines

scanner and parser is drafted.
1 never reduce.
34 shift/reduce conflicts
------------------------------------------------------------------------
r22 | marksun1988@gmail.com | 2011-10-25 23:05:53 -0400 (Tue, 25 Oct 2011) | 1 line

fix a problem of floats
------------------------------------------------------------------------
r21 | probas.322@gmail.com | 2011-10-24 17:22:50 -0400 (Mon, 24 Oct 2011) | 1 line

finished section 6 function and section 7 scope
------------------------------------------------------------------------
r20 | probas.322@gmail.com | 2011-10-24 14:00:26 -0400 (Mon, 24 Oct 2011) | 1 line

Section 5 Statement done.
------------------------------------------------------------------------
r19 | akash1531@gmail.com | 2011-10-23 23:39:33 -0400 (Sun, 23 Oct 2011) | 1 line

Added Initial ast.mli file similar to Micro C compiler
------------------------------------------------------------------------
r18 | marksun1988@gmail.com | 2011-10-22 22:42:28 -0400 (Sat, 22 Oct 2011) | 3 lines

1. code scanner.mll
2. add token types to parser.mly
3. fix tml_lrm.tex: change char size and bool size
------------------------------------------------------------------------
r17 | marksun1988@gmail.com | 2011-10-21 01:04:46 -0400 (Fri, 21 Oct 2011) | 1 line

add scanner.mll and parser.mly
------------------------------------------------------------------------
```

```
r16 | marksun1988@gmail.com | 2011-10-19 21:28:29 -0400 (Wed, 19 Oct 2011) | 1 line

revised at discussion
------------------------------------------------------------------------
r15 | marksun1988@gmail.com | 2011-10-19 04:39:15 -0400 (Wed, 19 Oct 2011) | 1 line

add constant definitions
------------------------------------------------------------------------
r14 | akash1531@gmail.com | 2011-10-19 03:56:57 -0400 (Wed, 19 Oct 2011) | 1 line

some additions to Lexical conventions
------------------------------------------------------------------------
r13 | marksun1988@gmail.com | 2011-10-19 03:27:41 -0400 (Wed, 19 Oct 2011) | 2 lines

conflict fixed.
This is a good version.
------------------------------------------------------------------------
r12 | marksun1988@gmail.com | 2011-10-19 03:21:20 -0400 (Wed, 19 Oct 2011) | 1 line

try to fix yan-akash-shuai conflict
------------------------------------------------------------------------
r11 | probas.322@gmail.com | 2011-10-18 21:03:04 -0400 (Tue, 18 Oct 2011) | 1 line


------------------------------------------------------------------------
r10 | akash1531@gmail.com | 2011-10-18 19:57:16 -0400 (Tue, 18 Oct 2011) | 1 line

restrictions
------------------------------------------------------------------------
r9 | akash1531@gmail.com | 2011-10-18 19:41:48 -0400 (Tue, 18 Oct 2011) | 1 line

Added more to introduction
------------------------------------------------------------------------
r8 | akash1531@gmail.com | 2011-10-18 19:23:23 -0400 (Tue, 18 Oct 2011) | 1 line

Added Introduction partially
------------------------------------------------------------------------
r7 | probas.322@gmail.com | 2011-10-18 16:18:39 -0400 (Tue, 18 Oct 2011) | 1 line

Test and add a new environment for typing codes
------------------------------------------------------------------------
r6 | hu.jiabin88@gmail.com | 2011-10-16 13:20:24 -0400 (Sun, 16 Oct 2011) | 2 lines

Expressions in LRM.

------------------------------------------------------------------------
r5 | marksun1988@gmail.com | 2011-10-12 20:45:59 -0400 (Wed, 12 Oct 2011) | 1 line


------------------------------------------------------------------------
r4 | marksun1988@gmail.com | 2011-10-12 20:42:55 -0400 (Wed, 12 Oct 2011) | 1 line

add the sections titles
------------------------------------------------------------------------
r3 | marksun1988@gmail.com | 2011-10-08 01:50:38 -0400 (Sat, 08 Oct 2011) | 1 line

add the tex file for tml lrm
------------------------------------------------------------------------
r2 | marksun1988@gmail.com | 2011-09-28 23:05:37 -0400 (Wed, 28 Sep 2011) | 1 line

ADD Docs directory and ADD two documents
------------------------------------------------------------------------
r1 | (no author) | 2011-09-21 18:26:27 -0400 (Wed, 21 Sep 2011) | 1 line

Initial directory structure.
------------------------------------------------------------------------
```

5 Architectural Design

This section presents the compiling and executing process of TML programs.

5.1 Compiling and Executing

TML compiler takes TML source code files with extension name .tml, as input. Firstly, the scanner scans the source code and break it into tokens. Then, the parser runs on the sequence of tokens and construct them into an abstract syntax tree(AST) according to the production rules. Taking the AST as the input, the analyzer check the types on it. It raises exceptions if there are type errors on the AST. Otherwise, the analyzer translate the AST to a "semantically-checked AST"(SAST). Besides type checking, the analyzer also computes some information, such as changing child aliases to child index number, and mark the information onto the SAST. By doing this, it helps the generator to generate codes. With this SAST, the generator then generates the byte code. The TML byte code is defined in bytecode.mli, and is based on stack. The generator outputs the byte code as text-format files with extension .tmb, which is the result of compiling.

The interpreter then takes the byte code file as input, run the instructions in it on stack, and output the result.

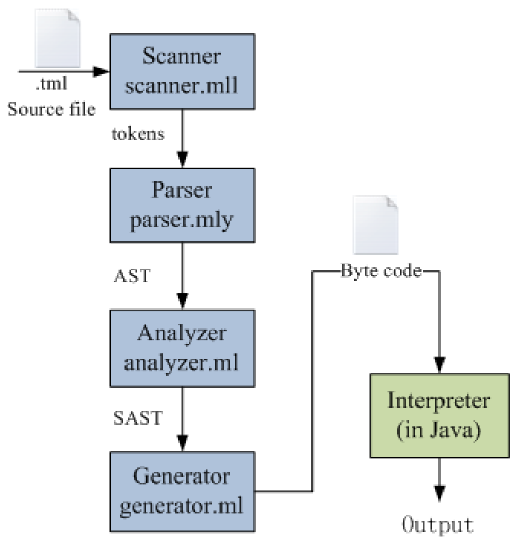

Figure 3: TML Compiling Process

The compiling and interpreting process is illustrated in Figure 3.

5.1.1 Interfaces between Components

The input of the compiler, which is first processed by the scanner is .tml source files. The scanner then generates a sequence of tokens, and the parser parses them into an AST. The analyzer translate the AST to SAST, which is further taken by the generator and turned into a text-format byte code file with extension .tmb. The .tmb byte code file is the output of the compiler and the input of the interpreter. The interpreter runs the code and gives output of the program.

5.1.2 Responsibility

Generally, everyone is well on the track of the project. The scanner and parser were implemented by Shuai. The definition of AST is written by Akash. The analyzer, generator and definition of SAST is written by Yan. The interpreter is implemented by Jiabin. For more details, please refer to the project log.

6 Test Plan

6.1 Representative Source

6.1.1 While Loop

- while.tml

```
void main()
{
        int i = 5;
        while (i < 10)
        {
                print(i, '\n');
                i = i + 1;
        }
}
```

- while.tmb

```
0       Glb 0
1       Jsr 3
2       Hlt
3       Ent 1
4       Psi 5
5       Bra 14
6       Lfp 1
7       Jsr -1
8       Pop 1
9       Psc 10
10      Jsr -1
11      Pop 1
12      Psi 0
13      Pop 1
14      Lfp 1
15      Psi 1
16      Add i
17      Sfp 1
18      Pop 1
19      Lfp 1
20      Psi 10
21      Lt i
22      Bne -16
23      Psi 0
24      Ret 0
```

6.1.2 Tree Connecting

- tt_connect.tml

```
treetype <2, [left, right]>MyTree_t
{
        int vint = 0;
        float vflt = 1.;
        string vstr = "hi";
}
```

```
void main ()
{
        MyTree_t ta, tb, tc;
        alloc(ta, tb, tc);

        ta.vstr = "root";
        tb.vstr = "left child";
        tc.vstr = "right child";

        ta ->(tb: tc);
        print (ta.vstr);                // root
        print (ta[left].vstr);          // left child
        print (ta[1].vstr);             // right child

}
```

- tt_connect.tmb

```
0       Glb 0
1       Jsr 3
2       Hlt
3       Ent 3
4       Pst
5       Pst
6       Pst
7       Lfp 1
8       Jsr 68
9       Sfp 1
10      Pop 1
11      Lfp 2
12      Jsr 68
13      Sfp 2
14      Pop 1
15      Lfp 3
16      Jsr 68
17      Sfp 3
18      Pop 1
19      Psi 0
20      Pop 1
21      Lfp 1
22      Pss 114 111 111 116
23      Sfd 2
24      Pop 1
25      Lfp 2
26      Pss 108 101 102 116 32 99 104 105 108 100
27      Sfd 2
28      Pop 1
29      Lfp 3
30      Pss 114 105 103 104 116 32 99 104 105 108 100
31      Sfd 2
32      Pop 1
33      Lfp 1
34      Psi 0
35      Lfp 2
36      Scd
```

```
37      Psi 1
38      Lfp 3
39      Scd
40      Pop 1
41      Lfp 1
42      Psi 2
43      Val T
44      Jsr -1
45      Pop 1
46      Psi 0
47      Pop 1
48      Lfp 1
49      Psi 0
50      Chd T
51      Psi 2
52      Val T
53      Jsr -1
54      Pop 1
55      Psi 0
56      Pop 1
57      Lfp 1
58      Psi 1
59      Chd T
60      Psi 2
61      Val T
62      Jsr -1
63      Pop 1
64      Psi 0
65      Pop 1
66      Psi 0
67      Ret 0
68      Ent 0
69      Lfp -2
70      Alc 2
71      Fld i
72      Fld f
73      Fld s
74      Sfp -2
75      Pop 1
76      Lfp -2
77      Psi 0
78      Sfd 0
79      Pop 1
80      Lfp -2
81      Psf 1.
82      Sfd 1
83      Pop 1
84      Lfp -2
85      Pss 104 105
86      Sfd 2
87      Pop 1
88      Lfp -2
89      Ret 1
```

6.1.3 Foreach Loop

- foreach.tml

```
treetype <2> MyTree_t
{
        int val = 0;
}


void main ()
{
        int i = 0;
        MyTree_t a, b, c, d, e, f;
        alloc(a, b, c, d, e, f);

        a -> (b -> (d : e -> (f : ~)): c);

        foreach node in a by preorder
        {
                node.val = i;
                i = i + 1;
        }

        foreach node in a by preorder
        {
                print (node.val);
                print (" ");
        }
        print ('\n');

        foreach node in a by inorder
        {
                print (node.val);
                print (" ");
        }
        print ('\n');

        foreach node in a by postorder
        {
                print (node.val);
                print (" ");
        }
        print ('\n');

        foreach node in a by levelorder
        {
                print (node.val);
                print (" ");
        }
}
```

- foreach.tmb

```
0       Glb 0
1       Jsr 3
2       Hlt
```

```
3        Ent 7
4        Psi 0
5        Pst
6        Pst
7        Pst
8        Pst
9        Pst
10       Pst
11       Lfp 2
12       Jsr 156
13       Sfp 2
14       Pop 1
15       Lfp 3
16       Jsr 156
17       Sfp 3
18       Pop 1
19       Lfp 4
20       Jsr 156
21       Sfp 4
22       Pop 1
23       Lfp 5
24       Jsr 156
25       Sfp 5
26       Pop 1
27       Lfp 6
28       Jsr 156
29       Sfp 6
30       Pop 1
31       Lfp 7
32       Jsr 156
33       Sfp 7
34       Pop 1
35       Psi 0
36       Pop 1
37       Lfp 2
38       Psi 0
39       Lfp 3
40       Psi 0
41       Lfp 5
42       Scd
43       Psi 1
44       Lfp 6
45       Psi 0
46       Lfp 7
47       Scd
48       Psi 1
49       Pst
50       Scd
51       Scd
52       Scd
53       Psi 1
54       Lfp 4
55       Scd
56       Pop 1
57       Lfp 2
58       Psi 0
```

```
59       Jsr -2
60       Lfp 8
61       Lfp 1
62       Sfd 0
63       Pop 1
64       Lfp 1
65       Psi 1
66       Add i
67       Sfp 1
68       Pop 1
69       Nxt -9
70       Pop 2
71       Lfp 2
72       Psi 0
73       Jsr -2
74       Lfp 8
75       Psi 0
76       Val T
77       Jsr -1
78       Pop 1
79       Psi 0
80       Pop 1
81       Pss 32
82       Jsr -1
83       Pop 1
84       Psi 0
85       Pop 1
86       Nxt -12
87       Pop 2
88       Psc 10
89       Jsr -1
90       Pop 1
91       Psi 0
92       Pop 1
93       Lfp 2
94       Psi 1
95       Jsr -2
96       Lfp 8
97       Psi 0
98       Val T
99       Jsr -1
100      Pop 1
101      Psi 0
102      Pop 1
103      Pss 32
104      Jsr -1
105      Pop 1
106      Psi 0
107      Pop 1
108      Nxt -12
109      Pop 2
110      Psc 10
111      Jsr -1
112      Pop 1
113      Psi 0
114      Pop 1
```

```
115     Lfp 2
116     Psi 2
117     Jsr -2
118     Lfp 8
119     Psi 0
120     Val T
121     Jsr -1
122     Pop 1
123     Psi 0
124     Pop 1
125     Pss 32
126     Jsr -1
127     Pop 1
128     Psi 0
129     Pop 1
130     Nxt -12
131     Pop 2
132     Psc 10
133     Jsr -1
134     Pop 1
135     Psi 0
136     Pop 1
137     Lfp 2
138     Psi 3
139     Jsr -2
140     Lfp 8
141     Psi 0
142     Val T
143     Jsr -1
144     Pop 1
145     Psi 0
146     Pop 1
147     Pss 32
148     Jsr -1
149     Pop 1
150     Psi 0
151     Pop 1
152     Nxt -12
153     Pop 2
154     Psi 0
155     Ret 0
156     Ent 0
157     Lfp -2
158     Alc 2
159     Fld i
160     Sfp -2
161     Pop 1
162     Lfp -2
163     Psi 0
164     Sfd 0
165     Pop 1
166     Lfp -2
167     Ret 1
```

6.2 Testing Details

Our project has a test suite with 43 test cases, along with expected output for each. Our test cases is written before we started coding semantic checking. We used the test cases to test scanner analyzer and the whole compiler, which ensures the correctness for every step forward.

Our test cases were written by all members, separating by parts. They covered primitive type operations, statement control flow, user defined tree type operations, and expressions. The test cases were as simple and independent as possible, so that we can identify where was the problem. Also, Akash added two complex TML programs. The test cases in the test suite are listed below:

./test/SrcCode/arithmetic_op.tml
./test/SrcCode/assignment_op.tml
./test/SrcCode/comparison_op.tml
./test/SrcCode/complex1.tml
./test/SrcCode/complex2.tml
./test/SrcCode/compound.tml
./test/SrcCode/continue.tml
./test/SrcCode/dangling_else.tml
./test/SrcCode/do_while.tml
./test/SrcCode/empty.tml
./test/SrcCode/empty_str.tml
./test/SrcCode/for.tml
./test/SrcCode/foreach.tml
./test/SrcCode/foreach_3nodes.tml
./test/SrcCode/foreach_single.tml
./test/SrcCode/function.tml
./test/SrcCode/function_forloop.tml
./test/SrcCode/gcd.tml
./test/SrcCode/global_var.tml
./test/SrcCode/if.tml
./test/SrcCode/logical_op.tml
./test/SrcCode/loop_control.tml
./test/SrcCode/main.tml
./test/SrcCode/print_backslash_chars.tml
./test/SrcCode/print_char.tml
./test/SrcCode/print_string.tml
./test/SrcCode/recursion.tml
./test/SrcCode/scope.tml
./test/SrcCode/tree_type1.tml
./test/SrcCode/tree_type2.tml
./test/SrcCode/tree_type3.tml
./test/SrcCode/tt_connect.tml
./test/SrcCode/tt_connect2.tml
./test/SrcCode/tt_data.tml
./test/SrcCode/tt_degree.tml

./test/SrcCode/tt_hash.tml
./test/SrcCode/tt_init_value.tml
./test/SrcCode/tt_init_value_nest.tml
./test/SrcCode/tt_node_copy.tml
./test/SrcCode/tt_parent.tml
./test/SrcCode/tt_tr_copy.tml
./test/SrcCode/var_init.tml
./test/SrcCode/while.tml

7 Lessons Learned

7.1 Jiabin Hu

Design is always much more difficult than implementation, or at least good design makes implementation trivial. With a good design, one just needs to translate the specification into code. With bad design, it's endless arguing and headache.

AST should be designed from bottom up, not top down. Slowly fill out the output of parser one by one starting from the most trivial construct, until the whole program can be represented.

Discover and handle the error as early as possible. If it can be detected in the scanner, don't wait until the parser.

Rome is not built in a day, nor is an compiler. Start from a minimum working compiler, and slowly add features to it. Run regression tests after every modification.

7.2 Akash Sharma

Using tools like Ocamlyacc and ocamllex speeds up writing a compiler

Version control was a good way to keep track of progress.

Paying attention to details while designing language, goes a long way while implementing it.

Regular team meetings were impetus to remain focused on task.

7.3 Shuai Sun

I was the team leader on this project. By "leader", it just means spending a little more time on planning. It was luck and amazing to have such teammates. We encountered problems and overcame them all along the semester. Finally, I learned a lot, which can be summarized as follows.

Compromising. As we never worked together before, we were different in working style and way of thinking. Sometimes, we ran into arguments. At this point, compromising was necessary to keep moving onto the next problem.

Planning. Our schedule was a little bit tight at the end of the semester, for members having other project dues. So, looking ahead for weeks even months was very necessary.

Getting into details. By this project, I got a more thoroughly understanding of how program codes are executed. But, before, when I was not willing to look deep into the code generation and executing, I just got to know the lexical, syntax and semantic parts.

In this project, we tried things that I never tried before, and they seemed to promote our work very well. One of them was writing test case suite in advance. Another was write a clear specification in advance. By doing them, we were totally aware of what we had accomplished and what was left to do.

7.4 Yan Zou

In this project, I mainly focused on how to implement semantic checking and code generating in OCaml, based on my own understanding of the knowledge got from this course. There are so many detailed things that we've went through in designing and implementing a compiler. I've learned a lot in each stage of the project.

Ideas. Brainstorm can bring about a lot of ideas, but it is difficult to choose. Everyone has his own thinking and concession is necessary. Voting is always good to try.

Building the LRM was the most difficult stage in this project, with no one clear about the direction. I realized that problems cannot simply get easier to solve when there are many people doing it, since we spent most of the time discussing and arguing and dealing with disagreements.

Coding can be done neatly by SVN. In this stage, I began to get a whole view of what a compiler is and how it works. By writing codes on every detail of a compiler, I came across the issues that professor has discussed during class one by one, like the order of parameter evaluation, scoping, and position of variable declaration, and so on.

Assembling and Testing. Test cases were designed before coding, so that we can test and debug all the way during coding. When we assembly all the parts, it simply works right. Above all, this project introduces me to the area of language design and implementation. It really teaches me a lot.

7.5 Advice on future teams

Finish scanner, parser, and define the AST before finalize on the LRM.

When you run into situations of endless argument, vote and let the leader make the call so that you can move on.

Work as a team. When some one is not able to accomplish job on time, give some direction or look into it with him/her.

Discussing is always making things clearer.

Subversion control and group emailing is necessary. Set them up at the very start.

8 Appendix

The directories in the following subsections are in alphabetical order.

8.1 trunk/src/TML_Compiler/

The following source code files are in alphabetical order.

8.1.1 analyzer_test.ml

```
(* analyzer_test.ml *)
(* @authors: Yan Zou *)

open Type
open Sast

let print_type = function
  | Int -> print_string "<int> "
  | Float -> print_string "<float> "
  | Char -> print_string "<char> "
  | String -> print_string "<string> "
  | Boolean -> print_string "<bool> "
  | Void -> print_string "<void> "
  | Tree_type tname -> print_string ("<tree:" ^ tname ^ "> ")

let print_order = function
  | Preorder -> print_string "preorder "
  | Inorder -> print_string "inorder "
  | Postorder -> print_string "postorder "
  | Levelorder -> print_string "levelorder "

let string_of_op = function
    Add -> "Add"
  | Sub -> "Sub"
  | Mult -> "Mul"
  | Div -> "Div"
  | Equal -> "Eq"
  | Neq -> "Neq"
  | Less_than -> "Lt"
  | Leq -> "Leq"
  | Greater_than -> "Gt"
  | Or -> "Or"
  | And -> "And"
  | Not -> "Not"
  | Geq -> "Geq"
  | Mod -> "Mod"
  | Dollar -> "Cln"
  | At -> "At"
  | Father -> "Fat"
  | Deg_a -> "Deg"
  | Dot -> "Val"
  | Hsh -> "Num"
  | Child -> "Chd"

let rec print_expr = function
  | Literal lit, t ->
      (match lit with
        | IntLit i -> print_int i
        | FloatLit f -> print_float f
        | CharLit c -> print_char c
        | BoolLit b -> print_string (string_of_bool b)
        | StringLit s -> print_string s
        | TreeLit -> print_char '~');
      print_type t
  | Id id, t ->
      print_string id; print_type t
  | Binop (e1, bin_op, e2), t ->
      print_expr e1; print_string (string_of_op bin_op);
      print_type t; print_expr e2
  | Assign (e1, e2), t ->
      print_expr e1; print_char '='; print_type t; print_expr e2
  | Call (func_name, params), t ->
      print_string func_name;
      print_type t;
      print_char '(';
      List.iter print_expr params;
      print_string ") ";
  | Uniop (un_op, e), t ->
      print_string (string_of_op un_op); print_type t; print_expr e;
  | Conn (e, el), t ->
      print_expr e; print_string "->"; print_type t; print_char '(';
      List.iter print_expr el; print_string ") "
  | Noexpr, t ->
      print_type t

let rec print_tab num =
  if (num > 0) then
    (print_char '\t'; print_tab (num - 1))
  else ()

let print_decl (t, name, init) =
  print_string ("declare: " ^ name); print_type t;
  match init with
    | None -> print_endline ";"
    | Some(e) ->
        print_string "= "; print_expr e; print_endline ";"

let print_local indent (t, name) =
  print_tab indent;
  print_string ("local: " ^ name);
  print_type t;
  print_endline ";"

let rec print_stmt indent s =
  print_tab indent; match s with
    | Block (sl, vl) ->
        print_endline "{";
        List.iter (print_local (indent + 1)) vl;
        List.iter (print_stmt (indent + 1)) sl;
        print_tab indent; print_endline "}"
    | Expr e -> print_expr e; print_endline ";"
    | Return e -> print_string "return "; print_expr e; print_endline ";"
    | If (e, s1, s2) ->
        print_string "if ("; print_expr e; print_endline ")";
        let new_indent = match s1 with
          | Block (_, _) -> indent
          | _ -> indent + 1
        in
        print_stmt new_indent s1;
        print_tab indent;
        print_endline "else";
        let new_indent = match s2 with
          | Block (_, _) -> indent
          | _ -> indent + 1
        in
        print_stmt new_indent s2
    | Foreach (id, e, order, s) ->
        print_string "foreach ("; print_string id; print_expr e;
        print_order order; print_endline ")";
        let new_indent = match s with
          | Block (_, _) -> indent
          | _ -> indent + 1
        in
        print_stmt new_indent s
    | For (e1, e2, e3, s) ->
        print_string "for (";
        print_expr e1; print_string "; ";
        print_expr e2; print_string "; ";
        print_expr e3; print_string "; ";
        print_endline ")";
        let new_indent = match s with
          | Block (_, _) -> indent
          | _ -> indent + 1
        in
        print_stmt new_indent s
    | Do (s, e) ->
        print_endline "do";
        let new_indent = match s with
          | Block (_, _) -> indent
          | _ -> indent + 1
        in
        print_stmt new_indent s;
        print_tab indent;
        print_string "while ("; print_expr e; print_endline ");";
    | While (e, s) ->
        print_string "while ("; print_expr e; print_endline ")";
        let new_indent = match s with
          | Block (_, _) -> indent
          | _ -> indent + 1
        in
        print_stmt new_indent s
    | Break i -> print_endline ("break " ^ (string_of_int i) ^ ";")
    | Continue i -> print_endline ("continue " ^ (string_of_int i) ^ ";")
    | Empty ->  print_endline ";"
    | Vardecl vd ->
        print_decl vd;;

let lexbuf = Lexing.from_channel stdin in
let program = Analyzer.check (Parser.program Scanner.token lexbuf) in
  List.iter print_decl program.globals;
  List.iter (fun decl ->
    print_string ("function: " ^ decl.fname);
    print_type decl.return_type;
    print_char '(';
    List.iter (fun (t, name) -> print_string name; print_type t) decl.params;
    print_endline ") ";
    print_stmt 0 (Block(decl.body, decl.locals))
  ) program.functions;
```

8.1.2 analyzer.ml

```
(* analyzer.ml *)
(* @authors: Yan Zou *)
(* @description: Semantic Analysis and convert AST to SAST *)

open Type
open Sast

module StringMap = Map.Make(String)

type tree_table = {
  type_name : string;
  degree : int; (* convert all degrees to numbers in analyzer *)
  aliases : int StringMap.t; (* convert all aliases to number *)
  member_list : (t * string) list
}

type symbol_table = {
  parent : symbol_table option;
  vars : (t * string) list;
  funcs : (t * string * (t list)) list;
  trees : tree_table list;
  child_enum : int StringMap.t;
  is_loop : bool (* true means this is a loop scope, false otherwise *)
  (* for break and continue to count the num of variables inside loop scope *)
}

let int_of_bool = function
```

```
    | true -> 1
    | false -> 0

let string_of_type = function
    | Int -> "integer"
    | Float -> "float"
    | Char -> "character"
    | String -> "string"
    | Boolean -> "boolean"
    | Tree_type tname -> ("Tree(" ^ tname ^ ")")
    | Void -> "void"

let string_of_op = function
      Add -> "+"
    | Sub -> "-"
    | Mult -> "*"
    | Div -> "/"
    | Mod -> "%"
    | Equal -> "=="
    | Neq -> "!="
    | Less_than -> "<"
    | Leq -> "<="
    | Greater_than -> ">"
    | Geq -> ">="
    | Or -> "||"
    | And -> "&&"
    | Not -> "!"
    | Dollar -> "$"
    | At -> "@"
    | Father -> "^"
    | Deg_a -> "&"
    | Dot -> "."
    | Hsh -> "#"
    | Child -> "[]"

(* check if two types are same *)
(* ~ should be regarded as the same type with all types of trees *)
let is_same_type t1 t2 =
  match (t1, t2) with
    | (Tree_type tt1, Tree_type tt2) ->
        if tt1 = "~" || tt2 = "~" then true
        else (tt1 = tt2)
    | (t1, t2) -> t1 = t2

(* find the varible in the scope by its name *)
let rec find_var scope name =
  try
    List.find (fun (_, var_name) -> var_name = name) scope.vars
  with Not_found ->
    match scope.parent with
      | Some(parent) ->find_var parent name
      | _ -> raise Not_found
(*
let rec find_func scope name =
  try
    List.find (fun (_, func_name, _) -> func_name = name) scope.func
  with Not_found -> (* for functions, we don't need to trace back to the root *)
    match scope.parent with
      | Some(parent) -> find_func parent name
      | _ -> raise Not_found *)

(* for trees, we don't need to trace back to the root *)
let find_tree scope name =
  try
    List.find (fun tree -> tree.type_name = name) scope.trees
  with Not_found ->
    raise (Failure ("Tree type " ^ name ^ " not declared"))

(* break the Ast.var_decl into lists of Sast.var_decl *)
let part decl =
    let t = fst decl in
    if t = Void then
      raise (Failure ("variables cannot be declared with type void"))
    else
      List.rev (List.fold_left (fun l -> function
        | Ast.WithInit (name, e) -> (t, name, Some(e))::l
        | Ast.WithoutInit name -> (t, name, None)::l) [] (snd decl))

(* operand type error *)
let e_op_type op t =
  raise (Failure ("The operand of operator " ^
    (string_of_op op) ^ " can not be " ^ (string_of_type t)))

(* convert Ast.expr to (Sast.expr, Type) by semantic analysis *)
let rec expr scope = function
  | Ast.Literal lit ->
      let lit_type = match lit with
        | IntLit i -> Int
        | FloatLit f -> Float
        | CharLit c -> Char
        | BoolLit b -> Boolean
        | StringLit s -> String
        | TreeLit -> Tree_type("~") (* universal tree type *)
      in
      Sast.Literal(lit), lit_type
  | Ast.Id id ->
      (try (* tree aliases first *)
        let i = StringMap.find id scope.child_enum in
          Sast.Literal(IntLit(i)), Int
      with Not_found ->
        try (* from local to global *)
          let (var_type, _) = find_var scope id in
            Sast.Id(id), var_type
        with Not_found ->
          raise (Failure ("undeclared identifier " ^ id)))
  | Ast.Binop (e1, bin_op, e2) ->
      let et1 = expr scope e1 in
      if bin_op = Dot then (* specially for tree operators *)
        match (snd et1) with
          | Tree_type tname ->
              let tree = find_tree scope tname in
              (* the expr following dot can only be ID of tree members *)
              let tree_scope = { scope with
                parent = None; vars = tree.member_list }
              in
              let et2 = expr tree_scope e2 in
              Sast.Binop(et1, bin_op, et2), (snd et2)
          | _ ->
            raise (Failure ("The left operand of operator . should be a tree type"))
      else if bin_op = Child then (* specially for tree operators *)
        match (snd et1) with
          | Tree_type tname ->
              let tree = find_tree scope tname in
              let tree_scope = { scope with child_enum = tree.aliases } in
              let et2 = expr tree_scope e2 in
              if (snd et2) = Int then
                Sast.Binop(et1, bin_op, et2), Tree_type(tname)
              else
                raise (Failure ("The expression inside [] should be an integer"))
              (*let et2 =
                try match e2 with
                  | Ast.Id id -> (* if it is an ID, search it in alias names *)
                      let i = StringMap.find id tree.aliases in
                      Sast.Literal(InitLit(i)), Int
                  | _ -> raise Not_found
                with Not_found -> (* not in alias names, evaluate it as usual *)
                  let et2 = expr scope e2 in
                  if (snd et2) = Int then et2
                  else
                    raise (Failure ("The expression inside [] should be an integer"))
              in Sast.Binop(et1, bin_op, et2), Tree_type(tname) *)
          | _ ->
            raise (Failure ("The left operand of operator [] should be a tree type"))
      else
        let et2 = expr scope e2 in
        let (_, t1) = et1 and (_, t2) = et2 in
         (*
        let e_right error =
          raise (Failure ("Right operand of operator " ^
            (string_of_op bin_op) ^ " can not be " ^ (string_of_type t2)))
        in
        let check_match = match bin_op with
          | Add | Sub | Mult | Div | Mod
          | Equal | Neq | Less_than | Leq | Greater_than | Geq ->
        in*)
        let t = (* operands should have the same type except tree operators *)
          if not (is_same_type t1 t2) then
              raise (Failure ("Type mismatch for operator " ^
                (string_of_op bin_op) ^ ": left - " ^ (string_of_type t1) ^
                                        ", right - " ^ (string_of_type t2)))
          else match bin_op with (* check operand type for different operators *)
            | Add -> (match t1 with
                  | Int | Float | String -> t1
                  | _ -> e_op_type bin_op t1)
            | Sub | Mult | Div -> (match t1 with
                  | Int | Float -> t1
                  | _ -> e_op_type bin_op t1)
            | Mod -> (match t1 with
                  | Int -> t1
                  | _ -> e_op_type bin_op t1)
            | Equal | Neq -> (match t1 with
                  | Int | Float | Char | String | Boolean
                  | Tree_type _ -> Boolean
                  | _ -> e_op_type bin_op t1)
            | Less_than | Leq | Greater_than | Geq -> (match t1 with
                  | Int | Float | Char | String -> Boolean
                  | _ -> e_op_type bin_op t1)
            | And | Or -> (if t1 = Boolean then Boolean else
              raise (Failure ("Only boolean is allowed for boolean operators")))
            | _ -> t1 (* TODO Tree operators *)
        in
        Sast.Binop(et1, bin_op, et2), t
  | Ast.Assign (e1, e2) ->
      let et1 = expr scope e1 and et2 = expr scope e2 in
      let (_, t1) = et1 and (_, t2) = et2 in
      if is_same_type t1 t2 then (* Type match *)
        Sast.Assign(et1, et2), t1
      else
        raise (Failure ("type mismatch in assignment"))
  | Ast.Call (func_name, params) ->
      let (func_type, _, required_param_types) = try
          List.find (fun (_, fname, _) -> fname = func_name) scope.funcs
        with Not_found ->
          raise (Failure ("undeclared identifier " ^ func_name))
      in
      let typed_params = List.map (expr scope) params in
      let param_types = List.map (fun et -> snd et) typed_params in
      let _ = match func_name with (* check parameter types *)
        | "print" -> List.iter (function
          | Tree_type _ -> (* print can only take in primitive types *)
              raise (Failure "print function can only take in primitive types")
          | _ -> ()) param_types
        | "alloc" -> List.iter (function
          | Tree_type _ -> ()
          | _ -> (* alloc can only take in tree types *)
              raise (Failure "alloc function can only take in tree types")
          ) param_types
        | _ -> if param_types <> required_param_types then (* other functions *)
            raise (Failure ("calling function " ^
                  func_name ^ " parameters mismatch"))
      in
      Sast.Call(func_name, typed_params), func_type
  | Ast.Uniop (un_op, e) ->
      let et = expr scope e in
      let t = snd et in
      if un_op = Deg_a then
        match t with
          | Tree_type tname ->
              let tree = find_tree scope tname in
              Sast.Literal(IntLit tree.degree), Int
          | _ ->
            raise (Failure ("Operator & takes only tree-typed operand"))
      else
        let tt = match un_op with
          | Add -> if t = Int or t = Float then t else
              raise (Failure ("Only integers and floats can be added a positive sign"))
          | Sub -> if t = Int or t = Float then t else
              raise (Failure ("Only integers and floats can be added a negative sign"))
          | Not -> if t = Boolean then Boolean else
              raise (Failure ("Only boolean is allowed for boolean operators"))
          | Dollar | At | Father | Hsh ->
              (match t with (* the existence of tree has been checked when declared *)
                | Tree_type tname -> if un_op = Hsh then Int else t
                | _ ->
                  raise (Failure ("Operator " ^ string_of_op un_op ^
                    " takes only tree-typed operand")))
          | _ -> raise (Failure ("The operator " ^
                  (string_of_op un_op) ^ " is not unary"))
        in
        Sast.Uniop(un_op, et), tt
  | Ast.Conn (e, el) ->
      let et = expr scope e and etl = List.map (expr scope) el in
      let t1 = snd et in
      (match t1 with
        | Tree_type tname ->
            let tree = find_tree scope tname in
            if (List.length etl) > tree.degree then
              raise (Failure ("Too many children in tree connection "^
                "for tree-type " ^ tname ^ " whose degree is only " ^
              (string_of_int tree.degree)))
            else
              let _ = List.iter (fun (_, t2) ->
                if (t2 <> Tree_type("~") && t2 <> t1) then
                  raise (Failure ("Tree type mismatch in tree connection"))
              ) etl
              in
              Sast.Conn(et, etl), t1
        | _ ->
          raise (Failure ("Operator & takes only tree-typed operand")))
  | Ast.Noexpr ->
      Sast.Noexpr, Void (* TODO: What is this? *)

(* convert Ast.expr to Sast.expr in the variable initialization *)
let check_init scope = function
  | Some e -> Some(expr scope e)
  | None -> None

(* count declared variables inside the current loop before break/continue *)
(* break/continue will pop out those variables from the stack before branching *)
let rec count_loop_vars scope num =
  if scope.is_loop then num
  else match scope.parent with
    | Some s -> count_loop_vars s (num + List.length scope.vars)
    | None -> raise (Failure "break/continue not in loops")

(* convert Ast.stmt to Sast.stmt *)
let rec stmt scope = function
  | Ast.Block sl ->
      (* create a new empty scope for the block, parent is the old scope *)
      let block_scope = { scope with (* func reserves *)
```

```
          parent = Some(scope); vars = []; is_loop = false}
        in
        (* check each statement in the block. scope may change in this process *)
        (* Vardecl and Foreach are dealt with here because they can add locals *)
        let check_stmt (block_scope, stl) s = match s with
          | Ast.Vardecl var_decl ->
              let _ = match (fst var_decl) with (* check tree type *)
                | Tree_type tname -> ignore(find_tree block_scope tname)
                | _ -> ()
              in
              let var_list = part var_decl in (* expand the multi-var declaration *)
              (* add all var in var_list to block_scope *)
              (* and convert the var_list(Ast.expr) to new_var_list(Sast.expr) *)
              let add_var (block_scope, new_var_list) (t, name, init) =
                try (* check if the variable has already been declared in this block *)
                  let _ =
                    List.find (fun (_, n) -> n = name) block_scope.vars
                  in raise (Failure ("variable " ^ name ^ " redeclared"))
                with Not_found -> (* no conflicts, simply continue *)
                  let new_scope = { (* add var to block_scope, inverse order *)
                    (* We don't need init info in block_scope *)
                    block_scope with vars = (t, name)::block_scope.vars
                  } and new_var_decl = (* convert init from Ast.expr to Sast.expr *)
                    (t, name, check_init scope init)
                  in (* add new_var_decl to new_var_list, inverse order *)
                  (new_scope, Sast.Vardecl(new_var_decl)::new_var_list)
              in
              let (new_scope, new_var_list) =
                List.fold_left add_var (block_scope, []) var_list
              in
              (new_scope, new_var_list @ stl) (* for one declaration *)
          | _ -> (block_scope, (stmt block_scope s)::stl)
        in
        let (block_scope, stl) =
          List.fold_left check_stmt (block_scope, []) sl
        in (* Note we don't have init info in block_scope *)
        (* The init is still in stmt list which can not be rearranged *)
        Sast.Block(List.rev stl, List.rev block_scope.vars)
    | Ast.Expr e -> Sast.Expr(expr scope e)
    | Ast.Return e -> Sast.Return(expr scope e)
    | Ast.ReturnVoid -> Sast.Return(Sast.Noexpr, Void)
    | Ast.If (e, s1, s2) ->
        let et = expr scope e in
        if ((snd et) = Boolean) then
          let st1 = stmt scope s1 in
          (* create a fake block for if statement to guarantee a new scope *)
          let new_st1 = match st1 with
            | Sast.Block(_, _) -> st1
            | _ -> stmt scope (Ast.Block [s1])
          in
          let st2 = stmt scope s2 in
          (* create a fake block for else clause to guarantee a new scope *)
          let new_st2 = match st2 with
            | Sast.Block(_, _) -> st2
            | _ -> stmt scope (Ast.Block [s2])
          in
          Sast.If(et, new_st1, new_st2)
        else
          raise (Failure ("the expression in if statement is not boolean"))
    | Ast.Foreach (id, e, order, s) ->
        let et = expr scope e in
        (match (snd et) with
          | Tree_type tname ->
              let tree = find_tree scope tname in (* check if the tree type exists *)
              if order = Inorder && tree.degree <> 2 then
                (* inorder only supports binary tree *)
                raise (Failure ("Foreach by inorder can only support binary trees"))
              else
                let loop_scope = (* add id to the current scope temporarily *)
                  { scope with
```

```
                    vars = (Tree_type(tname), id)::scope.vars;
                    is_loop = true }
                in
                let block_s = match s with
                  | Ast.Block _ -> s
                  | _ -> Ast.Block [s]
                in
                let st = stmt loop_scope block_s in
                (match st with (* check if id re-declared in the block *)
                  | Sast.Block(_, block_vars) ->
                      (try (* check if the id is re-declared inside foreach *)
                        let _ =
                          List.find (fun (t, name) -> name = id) block_vars
                        in raise (Failure ("Foreach variable " ^
                                              id ^ " redeclared"))
                      with Not_found -> (* no conflicts, convert it *)
                        Sast.Foreach (id, et, order, st))
                  | _ -> raise (Failure "unexpected failure for foreach block"))
          | _ -> raise (Failure ("identifier " ^ id ^ " is not a tree type")))
    | Ast.For (e1, e2, e3, s) ->
        let et2 = expr scope e2 in
        if ((snd et2) = Boolean) then
          let loop_scope = (* mark it as a loop *)
            { scope with is_loop = true }
          in
          (* create a fake block for for statement to guarantee a new scope *)
          let block_s = match s with
            | Ast.Block _ -> s
            | _ -> Ast.Block [s]
          in
          let st = stmt loop_scope block_s in
          Sast.For (expr scope e1, et2, expr scope e3, st)
        else
          raise (Failure ("the second expression in for statement is not boolean"))
    | Ast.Do (s, e) ->
        let et = expr scope e in
        if ((snd et) = Boolean) then
          let loop_scope = (* mark it as a loop *)
            { scope with is_loop = true }
          in
          (* create a fake block for for statement to guarantee a new scope *)
          let block_s = match s with
            | Ast.Block _ -> s
            | _ -> Ast.Block [s]
          in
          let st = stmt loop_scope block_s in
          Sast.Do (st, expr scope e)
        else
          raise (Failure ("the expression in do statement is not boolean"))
    | Ast.While (e, s) ->
        let et = expr scope e in
        if ((snd et) = Boolean) then
          let loop_scope = (* mark it as a loop *)
            { scope with is_loop = true }
          in
          (* create a fake block for for statement to guarantee a new scope *)
          let block_s = match s with
            | Ast.Block _ -> s
            | _ -> Ast.Block [s]
          in
          let st = stmt loop_scope block_s in
          Sast.While (expr scope e, st)
        else
          raise (Failure ("the expression in while statement is not boolean"))
    | Ast.Break -> Sast.Break (count_loop_vars scope 0)
    | Ast.Continue -> Sast.Continue (count_loop_vars scope 0)
    | Ast.Empty -> Sast.Empty
    | Ast.Vardecl _ -> (* handled in block for scope change *)
        raise (Failure ("Variable Declaration in wrong position!"))
        (* this can happen: if (a == 1) int b = 2; *)
```

```
let check program =
  let init_glob_scope = {
    parent = None;
    vars = [];
    funcs = [(Tree_type("~"), "alloc", []); (Void, "print", [])]; (* built-in functions
        *)
    trees = [];
    child_enum = StringMap.empty;
    is_loop = false;
  } in
  (* Global variable declarations and function definitions can be rearranged *)
  (* But local variable declarations can not be rearranged with other statements *)
  let (tree_list, glob_list, func_list) =
    (* expand multiple variable declarations into declaration lists *)
    let rec classify tree_list glob_list func_list glob_scope = function
      | [] -> (List.rev tree_list, glob_list, List.rev func_list)
      | (Ast.Globalvar var_decl)::tl -> (* put it into the loop *)
          let _ = match (fst var_decl) with (* check tree type *)
            | Tree_type tname -> ignore(find_tree glob_scope tname)
            | _ -> ()
          in
          let var_list = part var_decl in
          let new_var_list = (* convert Ast.expr to Sast.expr *)
            List.map (fun (t, name, init) ->
              try (* check if the variable has been declared before *)
                let _ =
                  List.find (fun (_, n) -> n = name) glob_scope.vars
                in raise (Failure ("global variable " ^ name ^ " redeclared"))
              with Not_found -> (* no conflicts, return the variable list *)
                (t, name, check_init glob_scope init)) var_list
          in
          let new_scope = { glob_scope with vars =
            glob_scope.vars @ (* add only type and name info into scope *)
            List.map (fun (vtype, name, _) -> (vtype, name)) var_list
          } in (* add new_var_list to glob_list in normal order *)
          classify tree_list (glob_list @ new_var_list) func_list new_scope tl
      | (Ast.Funcdef func_decl)::tl ->
          let ft = func_decl.Ast.return_type
            and fn = func_decl.Ast.fname
            and fp = func_decl.Ast.params
          in
          (try (* check if the function has been declared before *)
            let _ =
              List.find (fun (_, n, _) -> n = fn) glob_scope.funcs
            in raise (Failure ("function " ^ fn ^ " redeclared"))
          with Not_found -> (* no conflicts, begin to convert it *)
            (List.iter (fun (t, name) -> (* check params with same name *)
              if t = Void then
                raise (Failure ("type of parameter " ^ name ^ " cannot be void"))
              else
                let num = (* for each param, count the name in fp *)
                  List.fold_left (fun i (_, n) ->
                    if (n = name) then i + 1 else i) 0 fp
                in if (num > 1) then (* param name exists more than once *)
                  raise (Failure ("more than one parameter has the name " ^
                    name))) fp);
            let required_param_types = (* only type info of the params *)
              List.map (fun p -> fst p) fp
            in
            let new_scope = { glob_scope with funcs = (* add function *)
              (ft, fn, required_param_types)::glob_scope.funcs
            } in
            let param_scope = { new_scope with (* add params *)
              parent = Some(new_scope); vars = fp} (* is_loop is still false *)
            in
            let func_block =
              stmt param_scope (Ast.Block(func_decl.Ast.body))
            in
            let (new_body, local_var) = match func_block with
```

```
              | Sast.Block(new_body, local_var) ->
                  (* check if the parameters are redeclared in function body *)
                  List.iter (fun (_, pname) -> (* for each parameter *)
                    try
                      let _ = (* search the function local variable list *)
                        List.find (fun (_, n) -> n = pname) local_var
                      in raise (Failure ("parameter " ^ pname ^
                                " redeclared in function " ^ fn))
                    with Not_found -> () (* no conflicts, accept the body *)
                  ) fp; (new_body, local_var)
              | _ -> raise (Failure ("unexpected fatal error"))
            in
            let new_func_decl = {
              return_type = ft;
              fname = fn;
              params = fp;
              locals = local_var;
              (* Note we don't have init info in local_var *)
              (* The init is still in function body which can not be rearranged *)
              body = new_body;
            } in (* add new_func_decl to func_list in reverse order *)
            classify tree_list glob_list (new_func_decl::func_list) new_scope tl)
      | (Ast.Treedef tree_def)::tl ->
          let tn = tree_def.Ast.typename in
          (try (* check if the tree_typed has been declared before *)
            let _ =
              List.find (fun t -> t.type_name = tn) glob_scope.trees
            in raise (Failure ("tree type " ^ tn ^ " redeclared"))
          with Not_found -> (* no conflicts, simply continue *)
            let td = tree_def.Ast.degree in
            let ta = tree_def.Ast.aliases in
            if (td < List.length ta) then
              raise (Failure ("Too many alias names for tree type " ^ tn))
            else
              let (alias_map, _) =
                List.fold_left (fun (m, i) s ->
                  (StringMap.add s i m, i + 1))
                  (StringMap.empty, 0) ta
              in
              let tm =
                List.concat (List.map part tree_def.Ast.members)
              in
              let tree_table = {
                type_name = tn;
                degree = td;
                aliases = alias_map;
                member_list = List.map (fun (t, name, _) -> (t, name)) tm
              } in
              let new_scope = { glob_scope with trees =
                tree_table::glob_scope.trees }
              in
              let new_tm = (* convert Ast.expr to Sast.expr *)
                List.map (fun (t, name, init) ->
                  let _ = match t with (* recursive tree type in members *)
                    | Tree_type tname -> ignore(find_tree new_scope tname)
                    | _ -> ()
                  in (t, name, check_init glob_scope init)) tm
              in
              let new_tree_def = {
                typename = tn;
                members = new_tm;
                Sast.degree = td;
              } in
              classify (new_tree_def::tree_list) glob_list func_list new_scope tl)
    in classify [] [] [] init_glob_scope (List.rev program)
  in {
    treetypes = tree_list;
    globals = glob_list;
    functions = func_list
  }
```

8.1.3 ast.mli

```
(* ast.mli *)
(* @authors: Akash, Shuai Sun *)

open Type

type expr = (* Expressions *)
    Literal of literal (* 42 *)
  | Id of string (* foo *)
  | Binop of expr * op * expr (* a + b *)
  | Assign of expr * expr (* a = b *)
  | Call of string * (expr list) (* foo(1, 25) *)
  | Noexpr (* While() *)
  | Uniop of op * expr   (*for unary operators *)
  | Conn of expr * (expr list)

type init = WithInit of string * expr
      | WithoutInit of string

type init_list = init list

type var_decl = t * init_list

type tree_def = {
  typename: string;
  members : var_decl list;
  degree :int;
  aliases : string list;
}

type stmt = (* Statements  nothing *)
     Block of (stmt list)
   | Expr of expr   (*foo = bar + 3; *)
   | Return of expr (* return 42 also includes return function_name *)
   | ReturnVoid
   | If of expr * stmt * stmt (* if (foo == 42) {} else {} *)
   | Foreach of string * expr * traverse_order * stmt   (* for each loop *)
   | For of expr * expr * expr * stmt (* for loop *)
   | Do of stmt * expr   (*do while loop *)
   | While of expr * stmt (* while (i<10) { i = i + 1 } *)
   | Break  (* break *)
   | Continue  (* continue *)
   | Vardecl of var_decl
   | Empty

type func_decl = {
    return_type: t;
    fname : string;
    params : (t * string) list;
    body : stmt list;
}

type construct =
     Globalvar of var_decl
   | Funcdef of func_decl
   | Treedef of tree_def

type program = construct list
```

8.1.4 bytecode.mli

```
(* bytecode.mli *)
(* @authors: Yan Zou *)

type bstmt =
  | Glb of int (* Indicate the number of global variables *)
  | Psi of int (* Push an integer *)
  | Psf of float (* Push a floating number *)
  | Psc of char (* Push a cbaracter *)
```

```
    | Pss of string (* Push a string *)
    | Psb of bool (* Push a boolean *)
    | Pst (* Push a tree *)
  (*  | Pop (* Pop out a literal from the stack *)*)
    | Pop of int (* Pop out several elements from the stack *)
    | Uop of (Type.op * Type.t) (* Perform unary operation on top one element of stack *)
    | Bin of (Type.op * Type.t) (* Perform binary operation on top two elements of stack *)
    | Lod of int (* Fetch global variable *)
    | Str of int (* Store global variable *)
    | Lfp of int (* Load frame pointer relative *)
    | Sfp of int (* Store frame pointer relative *)
    | Jsr of int (* Call function by absolute address *)
    | Ent of int (* Entry of a function *)
    | Ret of int (* Restore FP, SP, consume formals, push result *)
    | Beq of int (* Branch relative if top-of-stack is zero *)
    | Bne of int (* Branch relative if top-of-stack is non-zero *)
    | Bra of int (* Branch relative *)
    | Alc of int (* new a tree for tree-typed variables, int is the degree *)
    | Fld of Type.t (* Add a new value field to a tree node *)
    | Sfd of int (* assign the value to the corresponding field of the tree *)
    | Scd (* assign the corresponding child of the tree, need three arguments *)
    | Nxt of int (* next iteration of foreach loop *)
    | Hlt (* Terminate *)
```

8.1.5 generator.ml

```
(* generator.ml *)
(* @authors: Yan Zou *)

open Bytecode
open Type
open Sast

module StringMap = Map.Make(String)

(* Index map for the three basic elements in TML *)
type environment = {
  func_index : int StringMap.t;
  global_index : int StringMap.t;
  tmember_index : int StringMap.t;
}

let string_of_type = function
  | Int -> "i"
  | Float -> "f"
  | Char -> "c"
  | String -> "s"
  | Boolean -> "b"
  | Tree_type tname -> "T"
  | Void -> "v"

let int_of_order = function
  | Preorder -> 0
  | Inorder -> 1
  | Postorder -> 2
  | Levelorder -> 3

let string_of_op = function
    Add -> "Add"
  | Sub -> "Sub"
  | Mult -> "Mul"
  | Div -> "Div"
  | Equal -> "Eq"
  | Neq -> "Neq"
  | Less_than -> "Lt"
  | Leq -> "Leq"
  | Greater_than -> "Gt"
  | Or -> "Or"
  | And -> "And"
  | Not -> "Not"
  | Geq -> "Geq"
  | Mod -> "Mod"
  | Dollar -> "Cln"
  | At -> "At"
  | Father -> "Fat"
  | Deg_a -> "Deg"
  | Dot -> "Val"
  | Hsh -> "Num"
  | Child -> "Chd"

(* convert a string to a list of ascii codes *)
let string_of_string s =
  let len = String.length s in
  let rec toasc result i =
    if i = len then result
    else
        let result =
          (result ^ " " ^ (string_of_int (Char.code (String.get s i))))
        in toasc result (i + 1)
  in toasc "" 0

(* convert bytecode to string *)
let string_of_bytecode = function
  | Glb i -> ("Glb " ^ (string_of_int i))
  | Psi i -> ("Psi " ^ (string_of_int i))
  | Psf f -> ("Psf " ^ (string_of_float f))
  | Psc c -> ("Psc " ^ (string_of_int (Char.code c)))
  | Pss s -> ("Pss" ^ (string_of_string s))
  | Psb b -> ("Psb " ^ (string_of_bool b))
  | Pst -> ("Pst")
  | Pop i -> ("Pop " ^ (string_of_int i))
  | Uop (un_op, t) ->
      ((if un_op = Sub then "Neg" else (string_of_op un_op)) ^
          " " ^ (string_of_type t))
  | Bin (bin_op, t) -> ((string_of_op bin_op) ^ " " ^ (string_of_type t))
  | Lod i -> ("Lod " ^ (string_of_int i))
  | Str i -> ("Str " ^ (string_of_int i))
  | Lfp i -> ("Lfp " ^ (string_of_int i))
  | Sfp i -> ("Sfp " ^ (string_of_int i))
  | Jsr i -> ("Jsr " ^ (string_of_int i))
  | Ent i -> ("Ent " ^ (string_of_int i))
  | Ret i -> ("Ret " ^ (string_of_int i))
  | Beq i -> ("Beq " ^ (string_of_int i))
  | Bne i -> ("Bne " ^ (string_of_int i))
  | Bra i -> ("Bra " ^ (string_of_int i))
  | Alc i -> ("Alc " ^ (string_of_int i))
  | Fld t -> ("Fld " ^ (string_of_type t))
  | Sfd i -> ("Sfd " ^ (string_of_int i))
  | Scd -> "Scd"
  | Nxt i -> ("Nxt " ^ (string_of_int i))
  | Hlt -> "Hlt"

let init_value t = function
  | Some(e) -> e
  | None ->
      let lit = match t with
        | Int -> IntLit(0)
        | Float -> FloatLit(0.0)
        | Char -> CharLit('\000')
        | String -> StringLit("")
        | Boolean -> BoolLit(false)
        | Tree_type tname -> TreeLit
        | Void -> IntLit(0) (* "Variables with type void shouldn't exist" *)
      in Literal(lit), t

(* add the names in the list to the map starting from start_index *)
let add_to_map list map start_index =
  snd (List.fold_left (fun (i, map) name ->
    (*print_endline (name ^ ": " ^ (string_of_int i)); (* debug *) *)
    (i + 1, StringMap.add name i map)) (start_index, map) list)
```

```
let translate out_filename program =
  (* index map for all globals - don't vary *)
  let glob_index =
    let gname_list =
      List.map (fun (_, n, _) -> n) program.globals
    in
    add_to_map gname_list StringMap.empty 0
  in
  (* index map for all functions - don't vary *)
  let func_index =
    let fname_list =
      List.map (fun f -> f.fname) program.functions
    in
    (* tree allocation function names are same with the tree type names *)
    (* after type checking, they can't be the same with other functions *)
    let tname_list =
      List.map (fun t -> t.typename) program.treetypes
    in
    let built_in = (* built-in functions *)
      add_to_map ["print"] StringMap.empty (-1)
    in
    let func_index_without_trees =
      add_to_map fname_list built_in 0
    in (* append tree alloc functions to the end of funcs *)
    add_to_map tname_list func_index_without_trees
      (List.length fname_list) (* starting offset of tree alloc functions *)
  in
  let tmember_index =
    let add_to_map_2 map t = (* for trees *)
      let tree_name = t.typename in
      let member_list = (* add tree_name before member names *)
        (* this avoids conflicts because "." is not allowed in identifiers *)
        List.map (fun (_, name, _) -> tree_name ^ "." ^ name) t.members
      in
      add_to_map member_list map 0
    in
    List.fold_left add_to_map_2 StringMap.empty program.treetypes
  in
  (* translate all the stmts in a block, local_index will not vary *)
  let rec block local_index next_index num_params sl =
    (* common function for Binop and Assign in generating expr *)
    let get_tmember_index e1 e2 =
      match (snd e1) with (* get tree type name *)
        | Tree_type tree_name ->
            (match e2 with
              | Id id, _ -> (* tree_name.id *)
                  (let member_id = tree_name ^ "." ^ id in
                  try StringMap.find member_id tmember_index
                  with Not_found ->
                    raise (Failure ("tree member " ^
                      member_id ^ " not found")))
              | _ ->
                  raise (Failure ("The right operand of operator " ^
                    ". can only be an identifier")))
        | _ ->
            raise (Failure ("The left operand of operator " ^
                    ". can only be tree type"))
    in
    (* Evaluate expression, the result is on top of the stack *)
    let rec expr = function
      | Literal lit, _ -> ( match lit with
        | IntLit i -> [Psi i]
        | FloatLit f -> [Psf f]
        | CharLit c -> [Psc c]
        | BoolLit b -> [Psb b]
        | StringLit s -> [Pss s]
        | TreeLit -> [Pst] )
      | Id id, _ -> (* the ID of tree member won't get here *)
          ( try [Lfp (StringMap.find id local_index)]
            with Not_found ->
              try [Lod (StringMap.find id glob_index)]
              with Not_found ->
                raise (Failure ("Variable " ^ id ^ " not found")))
      | Binop (e1, bin_op, e2), t ->
          let be2 =
            if bin_op = Dot then
              [Psi (get_tmember_index e1 e2)]
            else expr e2
          in
          (expr e1) @ be2 @ [Bin (bin_op, (snd e1))]
      | Assign (e1, e2), _ ->
          (* left value for assignment, get the address or use Sfp/Str/Sfd/Scd *)
          (match e1 with
            | Id id, t ->
                let lvalue_bytecode =
                  ( try [Sfp (StringMap.find id local_index)]
                    with Not_found ->
                      try [Str (StringMap.find id glob_index)]
                      with Not_found ->
                        raise (Failure ("Variable " ^ id ^ " not found")))
                in (expr e2) @ lvalue_bytecode
            | Binop (e11, bin_op, e12), t ->
                ( match bin_op with
                    | Dot -> let i = get_tmember_index e11 e12 in
                        (expr e11) @ (expr e2) @ [Sfd i]
                    | Child ->
                        (expr e11) @ (expr e12) @ (expr e2) @ [Scd]
                    | _ -> raise (Failure "Illegal left value") )
            | _ -> raise (Failure "Illegal left value"))
      | Call (func_name, params), t ->
          if (func_name = "print") then
            if (List.length params > 0) then
              (* expr (List.hd params) @ [Jsr (-1)] @
              List.concat (List.map (fun e -> [Pop 1] @ (expr e) @
                                        [Jsr (-1)]) (List.tl params))*)
              (List.concat (List.map (fun e -> (* print one by one *)
                (expr e) @ [Jsr (-1); Pop 1]) params)) @
              [Psi 0] (* return value of print *)
            else []
          else if (func_name = "alloc") then
            let alloc e = (* alloc one tree *)
              match e with (* alloc only when the parameters are tree-type left values *)
                | Id _, Tree_type tree_name
                | Binop (_, Dot, _), Tree_type tree_name
                | Binop (_, Child, _), Tree_type tree_name ->
                    let ea =
                      Assign (e, (Call (tree_name, [e]),
                        Tree_type tree_name)), Tree_type tree_name
                    in
                    expr ea @ [Pop 1]
                | _ -> (* else, do nothing *)
                    raise (Failure "illegal alloc parameters")
            in
            if (List.length params > 0) then (* alloc one by one *)
              (List.concat (List.map (fun e -> alloc e) params)) @
              [Psi 0] (* void return of alloc *)
            else []
          else
            List.concat (List.map expr params) @
            (try [Jsr (StringMap.find func_name func_index)]
             with Not_found ->
              raise (Failure ("Function " ^ func_name ^ " not found")))
      | Uniop (un_op, e), t ->
          if un_op = Add then (expr e) else ((expr e) @ [Uop (un_op, t)])
      | Conn (e, el), _ -> (* add to children one by one *)
          let (_, add_children_byte) =
            List.fold_left (fun (i, bl) ec ->
              (i + 1, bl @ [Psi i] @ (expr ec) @ [Scd])) (0, []) el
          in
```

```
          expr e @ add_children_byte
      | Noexpr, _ -> [Psi 0] (* void return of a function *)
    in
    let loop_control sl continue_offset break_offset =
      let (_, new_sl) =
        List.fold_left (fun (i, l) -> function
          (* the Jsr -3 and -4 here are just stubs which will be changed later *)
          | Jsr (-3) -> (i + 1, (Bra (break_offset - i))::l) (* Break *)
          | Jsr (-4) -> (i + 1, (Bra (continue_offset - i))::l) (* Continue *)
          | _ as s -> (i + 1, s::l)) (0, []) sl
      in List.rev new_sl
    in
    let rec stmt = function
      | Block (sl, vl) -> (* need new locals *)
          let vname_list = List.map (fun (_, n) -> n) vl in
          let new_local_index =
            add_to_map vname_list local_index next_index
          in
          let num_locals = List.length vl in
          (block new_local_index (next_index + num_locals) num_params sl) @
          ( if num_locals > 0 then
              [Pop num_locals] (* pop out the locals in the block to save space *)
            else [] )
      | Expr e -> expr e @ [Pop 1]
      | Return e -> expr e @ [Ret num_params]
      | If (e, s1, s2) ->
          let r1 = (stmt s1) and r2 = (stmt s2) in
          expr e @ [Beq ((List.length r1) + 2)] @ r1 @
          [Bra ((List.length r2) + 1)] @ r2
      | Foreach (id, e, order, s) ->
          let new_local_index =
            (* print_endline (id ^ ": " ^ (string_of_int next_index)); (* debug *)*)
            StringMap.add id next_index local_index
          in
          let rs = (* regard the foreach statement as a block *)
            (* the id can only be used inside foreach *)
            (* index moves 2 forward because of id and order *)
            block new_local_index (next_index + 2) num_params [s] (* double blocks *)
          in
          let continue_offset = List.length rs in
          let break_offset = continue_offset + 1 in
          expr e @ (* push the root node onto the stack *)
          [Psi (int_of_order order)] @ (* traverse order *)
          (* [Psi (List.length bytecodes)] @ (* number of lines in foreach *) *)
          [Jsr (-2)] @
          (loop_control rs continue_offset break_offset) @
          [Nxt (- List.length rs)] @ [Pop 2]
          (* in order to make variables in the right order on the stack *)
      | For (e1, e2, e3, s) ->
          (* stmt (Block([Expr(e1); While(e2, Block([s; Expr(e3)]))])) *)
          let re1 = stmt (Expr e1) and re2 = expr e2 and
              re3 = stmt (Expr e3) and rs = stmt s
          in
          let continue_offset = List.length rs in
          let check_offset = continue_offset + (List.length re3) in
          let break_offset = check_offset + (List.length re2) + 1 in
          re1 @ [Bra (check_offset + 1)] @
          (loop_control rs continue_offset break_offset) @
          re3 @ re2 @ [Bne (-break_offset + 1)]
      | Do (s, e) ->
          let rs = stmt s and re = expr e in
          let continue_offset = List.length rs in
          let break_offset = continue_offset + (List.length re) + 1 in
          (loop_control rs continue_offset break_offset) @
          re @ [Bne (-break_offset + 1)]
      | While (e, s) ->
          let re = expr e and rs = stmt s in
          let continue_offset = List.length rs in
          let break_offset = continue_offset + (List.length re) + 1 in
          [Bra (continue_offset + 1)] @
          (loop_control rs continue_offset break_offset) @
          re @ [Bne (-break_offset + 1)]
      (* break and continue are just stubs *)
      (* in case that local variables are declared in the loop scope *)
      (* we need to record the current number of local variables in the stub *)
      (* they will be adjusted later so that break or continue *)
      (* will pop out all local variables in the loop scope before branching *)
      | Break i -> [Pop i; Jsr (-3)] (* stub, Indicate it's a break *)
      | Continue i -> [Pop i; Jsr (-4)] (* Indicate it's a continue *)
      | Vardecl (t, name, init) -> expr (init_value t init) (* leave it on the stack *)
        (* The stack space allocated for the new locals move along with the sp *)
        (* The locals will be consecutive in space as var_decl is a complete *)
        (* statement. There shouldn't be any intermediate result on the stack *)
        (* between statements, which only appears when evaluating expressions. *)
      | Empty -> []
    in
    List.concat (List.map stmt sl)
  in
  let data =
    (* convert all the global declarations into a block of assignments *)
    let assign_globals =
      List.map (fun (t, name, init) ->
        Expr(Assign((Id(name), t), init_value t init), t)) program.globals
    in
    [Glb (List.length assign_globals)] @
    block StringMap.empty 0 0 assign_globals @
    try
      [Jsr (StringMap.find "main" func_index); Hlt]
    with Not_found ->
      raise (Failure "main function not found!")
  in
  let text =
    (* convert function bodies into bytecodes *)
    let trans_func f =
      let param_names = List.map snd f.params in (* get the names *)
      let num_params = List.length param_names in
      let param_index = (* the last parameter will have index -2 *)
        add_to_map param_names StringMap.empty (-1 - num_params)
      in
      let local_names = List.map snd f.locals in (* get the names *)
      let num_locals = List.length local_names in
      let local_index = (* the first local variable will have index 1 *)
        add_to_map local_names param_index 1
      in
      let bodycodes =
        block local_index (num_locals + 1) num_params f.body
      in
      (* check if there are break or continue statements not adjusted *)
      List.iter (function
        | Jsr (-3) -> raise (Failure "break not in loops")
        | Jsr (-4) -> raise (Failure "continue not in loops")
        | b -> ()) bodycodes;
      [Ent num_locals] @ bodycodes @
      (match f.return_type with
        | Int -> [Psi 0]
        | Float -> [Psf 0.0]
        | Char -> [Psc '\000']
        | String -> [Pss ""]
        | Boolean -> [Psb false]
        | Tree_type tname -> [Pst]
        | Void -> [Psi 0]) @
      [Ret num_params]
    in
    (* convert tree definitions into alloc functions *)
    let trans_trees treetype =
      let init_members = (* generate the member initialization statements *)
        List.map (fun (t, name, init) ->
          Expr(Assign((Binop((Id("."), Tree_type(treetype.typename)),
            Dot, (Id(name), t)), t), init_value t init), t))
          treetype.members
      in
      let tree_init_map = (* fake map only containing the tree itself *)
        StringMap.add "." (-2) StringMap.empty
      in
      [Ent 0; Lfp (-2); Alc treetype.degree] @ (* get the tree - the only parameter *)
      List.map (fun (t, _, _) -> Fld t) treetype.members @ [Sfp (-2); Pop 1] @
      block tree_init_map 0 1 init_members @
      [Lfp (-2); Ret 1] (* return the tree *)
    in
    let func_bodies =
      (List.map trans_func program.functions) @
      (List.map trans_trees program.treetypes)
    in
    let (_, func_offsets) = (* generate the function offsets as a list *)
        List.fold_left (fun (i, offsets) body ->
            (* each step plus the number of byte statements *)
            ((List.length body) + i, i::offsets))
          (* start from the end of global variable initialization *)
          ((List.length data), []) func_bodies
          (* +1 is for Glb *)
    in
    let array_offsets = Array.of_list (List.rev func_offsets) in
    List.map (function
      | Jsr i when i >= 0 -> Jsr array_offsets.(i)
      | _ as s -> s)
      (List.concat (data::func_bodies))
  in
  let fout = open_out out_filename in
    ignore ( List.fold_left (fun i b ->
        (* output line number for debug *)
        output_string fout ((string_of_int i) ^ "\t" ^ (string_of_bytecode b) ^ "\n");
        (i + 1)) 0 text
    );
    close_out fout
```

8.1.6 parser.mly

```
/* parser.mly */
/* @authors: Shuai Sun, Yan Zou, Jiabin Hu, Akash */
%{ open Type %}
%{ open Ast %}

%token <string> ID
%token IF ELSE WHILE DO FOR BREAK CONTINUE FOREACH IN BY RETURN
%token PREORDER INORDER POSTORDER LEVELORDER
%token INT_T FLOAT_T CHAR_T STRING_T BOOL_T VOID TREETYPE
%token <int>INT
%token <float>FLOAT
%token <bool>BOOL
%token <string>STRING
%token <char>CHAR
%token NULL
%token LBRACE RBRACE SEMI COLON COMMA
%token ASSIGN
%token CONNECT
/* These operators are removed
%token PLUS_ASN MINUS_ASN TIMES_ASN DIVIDE_ASN MOD_ASN
*/
%token OR AND NOT
%token NEQ GT LT LEQ GEQ EQ
%token PLUS MINUS TIMES DIVIDE MOD
%token DOLLAR AT LBRACK RBRACK DEG_AND DOT HASH FATHER
%token LPAREN RPAREN
%token EOF

%nonassoc NOELSE
%nonassoc ELSE

%right CONNECT
%right ASSIGN
```

```
/* These operators are removed
%right PLUS_ASN MINUS_ASN
%right TIMES_ASN DEVIDE_ASN MOD_ASN
*/
%left OR
%left AND
%left EQ NEQ
%left LT GT LEQ GEQ
%left PLUS MINUS   /* for binop they are left, but for unop should they be right? */
%right SIGN
%left TIMES DIVIDE MOD
%right NOT
%right AT, DOLLAR, FATHER
%right HASH, DEG_AND
%nonassoc LBRACK
%left DOT

%start program
%type <Ast.program> program      /* this type should be AST.program. just put int
    because AST is not finished */
%%


program:
    /* nothing */                             { [] }   /* this part should be differnt
                                                        and it should be similar to
                                                            MicroC unsure about it */
    | program type_def                        { Treedef($2)::$1 }
    | program decl                            { Globalvar($2)::$1 }
    | program func_def                        { Funcdef($2)::$1 }

type_def:
    TREETYPE LT INT GT ID LBRACE decl_list RBRACE
                                                            { { typename = $5;
                                                                members = List.rev $7;
                                                                degree = $3;
                                                                aliases = [] }}
    | TREETYPE LT INT COMMA LBRACK alias_list RBRACK GT ID LBRACE decl_list RBRACE
                                                            { { typename = $9;
                                                                members = List.rev $11;
                                                                degree = $3;
                                                                aliases = List.rev $6 } }

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

decl_list:
    decl                                    { [$1] }
    | decl_list decl                        { $2::$1 }

decl:
    type_specifier init_list SEMI           { ($1, List.rev $2) }

type_specifier:
    INT_T                                  { Int }
    | FLOAT_T                              { Float }
    | CHAR_T                               { Char }
    | STRING_T                             { String }
    | BOOL_T                               { Boolean }
    | ID                                   { Tree_type($1) }
    | VOID                                 { Void }

/*
return_type:
    type_specifier                         { $1 }
    | VOID                                 { Void }
*/

init_list:
    init                                    { [$1] }
    | init_list COMMA init                  { $3::$1 }

init:
    ID                                      { WithoutInit($1) }
    | ID ASSIGN expr                        { WithInit($1 ,$3) }

func_def:
    type_specifier ID LPAREN para_list RPAREN stmt_block           { {  return_type = $1;
                                                                        fname = $2;
                                                                        params =
                                                                            List.rev $4;
                                    body = $6
                                  } }

para_list:
    /* nothing */                  {[]}
    | para_decl                             { [$1] }
    | para_list COMMA para_decl             { $3::$1 }

para_decl:
    type_specifier ID                       { ($1, $2) }

stmt_block:
    LBRACE stmt_list RBRACE                 { List.rev $2 }

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

stmt:
    expr SEMI                                         { Expr($1) }
    | decl                                                    { Vardecl($1) }
    | stmt_block                                              { Block($1) }
    | IF LPAREN expr RPAREN stmt %prec NOELSE                 { If($3, $5, Empty) }
    | IF LPAREN expr RPAREN stmt ELSE stmt                    { If($3, $5, $7) }
    | WHILE LPAREN expr RPAREN stmt                           { While($3, $5) }
    | DO stmt WHILE LPAREN expr RPAREN SEMI                   { Do($2, $5) }
    | FOR LPAREN expr SEMI expr SEMI expr RPAREN stmt         { For($3, $5, $7, $9) }
    /*| FOREACH LPAREN ID IN expr BY trvs_order RPAREN stmt     { Foreach($3, $5, $7,
        $9) } */
    | FOREACH ID IN expr BY trvs_order stmt                   { Foreach($2, $4, $6, $7) }
                                                                  /* drop the
                                                                      parenthesis
                                                                      */
    | BREAK SEMI                                              { Break }
    | CONTINUE SEMI                                           { Continue }
    | RETURN expr SEMI                                        { Return($2) }
    | RETURN SEMI                                             { ReturnVoid }
    | SEMI                                                    { Empty }      /*No action
        really.. ?*/

trvs_order:
    INORDER                                               { Inorder }
                                                            /*I doubt what exactly these
                                                                should contain..?*/
    | PREORDER                                            { Preorder }
    | POSTORDER                                           { Postorder }
    | LEVELORDER                                          { Levelorder }

expr:
    | literal                             { Literal($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 MOD expr                       { Binop($1, Mod, $3) }

    | expr GT expr                        { Binop($1, Greater_than, $3) }
    | expr LT expr                         { Binop($1, Less_than, $3) }
    | expr GEQ expr                        { Binop($1, Geq, $3) }
    | expr LEQ expr                        { Binop($1, Leq, $3) }
    | expr NEQ expr                        { Binop($1, Neq, $3) }
    | expr EQ expr                         { Binop($1, Equal, $3) }

    | expr AND expr                        { Binop($1, And, $3) }
    | expr OR expr                         { Binop($1, Or, $3) }

    | PLUS expr                            { Uniop(Add, $2) }
    | MINUS expr                           { Uniop(Sub, $2 ) }
    | AT expr                          { Uniop(At, $2) }
    | DOLLAR expr                          { Uniop(Dollar, $2) }
    | FATHER expr                          { Uniop(Father, $2) }
    | NOT expr                             { Uniop(Not, $2) }
    | HASH expr                            { Uniop(Hsh, $2) }
    | DEG_AND expr                         { Uniop(Deg_a, $2) }

    | lvalue                               { $1 }

    | lvalue ASSIGN expr                   { Assign($1, $3) }

    | lvalue CONNECT LPAREN node_list RPAREN  { Conn($1, List.rev  $4) }

    | LPAREN expr RPAREN                   { $2 }
    | ID LPAREN arg_list RPAREN            { Call($1, List.rev $3) }


literal:
    INT                                    { IntLit($1) }
    | FLOAT                                { FloatLit($1) }
    | STRING                               { StringLit($1) }
    | CHAR                                 { CharLit($1) }
    | BOOL                                 { BoolLit($1) }
    | NULL                                 { TreeLit }
                                             /*We dont have a NULL type defined in ast
                                                 yet*/

node_list:
    expr                                   { [$1] }
    | node_list COLON expr                 { $3::$1 }

lvalue:
    ID                                     { Id($1) }
    | expr DOT ID                          { Binop($1, Dot, Id($3)) }
    | expr LBRACK expr RBRACK              { Binop($1, Child, $3) }

arg_list:
    /* nothing */                          { [] }
    | expr                                 { [$1] }
    | arg_list COMMA expr                  { $3::$1 }
```

8.1.7 sast.mli

```
(* sast.mli *)
(* @authors: Yan Zou *)

open Type

type expr_c = (* Expressions *)
    Literal of literal (* 42 *)
  | Id of string (* foo *)
  | Binop of expr * op * expr (* a + b *)
  | Assign of expr * expr (* a = b *)
  | Call of string * (expr list) (* foo(1, 25) *)
  | Noexpr (* While() *)
  | Uniop of op * expr  (*for unary operators *)
  | Conn of expr * (expr list)
and expr = expr_c * t
```

```
type var_decl = t * string * (expr option)

type tree_def = {
  typename: string;
  members : var_decl list;
  degree : int;
}

type stmt = (* Statements  nothing *)
     Block of (stmt list) * ((t * string) list) (* statement list and var list *)
   | Expr of expr   (* foo = bar + 3; *)
   | Return of expr (* return 42 also includes return function_name *)
   | If of expr * stmt * stmt (* if (foo == 42) {} else {} *)
   | Foreach of string * expr * traverse_order * stmt   (* for each loop *)
   | For of expr * expr * expr * stmt (* for loop *)
   | Do of stmt * expr   (* do while loop *)
   | While of expr * stmt (* while (i<10) { i = i + 1 } *)
   | Break of int (* break, followed by the num of loop-scope locals need to be popped*)
   | Continue of int (* continue, followed by the num of loop-scope locals need to be
       popped *)
   | Vardecl of var_decl
   | Empty

type func_decl = {
    return_type : t;
    fname : string;
    params : (t * string) list;
    locals : (t * string) list; (* we don't put init here, it's in body *)
    body : stmt list; (* variable init is a stmt that can not be rearranged *)
}

type program = {
    treetypes: tree_def list;
    globals: var_decl list;
    functions: func_decl list;
}
```

8.1.8 scanner_test.ml

```
(* scanner_test.ml *)
(* @author: Yan Zou *)
(* To test the scanner by outputting all the recognized lexical words *)

open Parser;;

let lexbuf = Lexing.from_channel stdin in
let wordlist =
let rec next l =
  let word =
    match Scanner.token lexbuf with
      | EOF -> "EOF"
      | ID(id) -> "ID(" ^ id ^ ")"
      | IF -> "IF"
      | ELSE -> "ELSE"
      | WHILE -> "WHILE"
      | DO -> "DO"
      | FOR -> "FOR"
      | BREAK -> "BREAK"
      | CONTINUE -> "CONTINUE"
      | FOREACH -> "FOREACH"
      | IN -> "IN"
      | BY -> "BY"
      | RETURN -> "RETURN"
      | PREORDER -> "PREORDER"
      | INORDER -> "INORDER"
      | POSTORDER -> "POSTORDER"
      | LEVELORDER -> "LEVELORDER"
      | INT_T -> "INT_T"
      | FLOAT_T -> "FLOAT_T"
      | CHAR_T -> "CHAR_T"
```

```
        | STRING_T -> "STRING_T"
        | BOOL_T -> "BOOL_T"
        | VOID -> "VOID"
        | TREETYPE -> "TREETYPE"
        | INT(i) -> "INT(" ^ (string_of_int i) ^ ")"
        | FLOAT(f) -> "FLOAT(" ^ (string_of_float f) ^ ")"
        | BOOL(b) -> "BOOL(" ^ (string_of_bool b) ^ ")"
        | STRING(s) -> "STRING(" ^ s ^ ")"
        | CHAR(c) -> "CHAR(" ^ (String.make 1 c) ^ ")"
        | NULL -> "NULL"
        | LBRACE -> "LBRACE"
        | RBRACE -> "RBRACE"
        | SEMI -> "SEMI"
        | COLON -> "COLON"
        | COMMA -> "COMMA"
        | ASSIGN -> "ASSIGN"
        | CONNECT -> "CONNECT"
        (* These operators are removed.
            | PLUS_ASN -> "PLUS_ASN"
        | MINUS_ASN -> "MINUS_ASN"
        | TIMES_ASN -> "TIMES_ASN"
        | DIVIDE_ASN -> "DIVIDE_ASN"
        | MOD_ASN -> "MOD_ASN"
            *)
        | OR -> "OR"
        | AND -> "AND"
        | NOT -> "NOT"
        | NEQ -> "NEQ"
        | GT -> "GT"
        | LT -> "LT"
        | LEQ -> "LEQ"
        | GEQ -> "GEQ"
        | EQ -> "EQ"
        | PLUS -> "PLUS"
        | MINUS -> "MINUS"
        | TIMES -> "TIMES"
        | DIVIDE -> "DIVIDE"
        | MOD -> "MOD"
        | DOLLAR -> "DOLLAR"
        | AT -> "AT"
        | LBRACK -> "LBRACK"
        | RBRACK -> "RBRACK"
        | DEG_AND -> "DEG_AND"
        | DOT -> "DOT"
        | HASH -> "HASH"
        | FATHER -> "FATHER"
        | LPAREN -> "LPAREN"
        | RPAREN -> "RPAREN"
        (*| _ -> "UNKNOWN"*)
      in if word = "EOF" then l
          else next (word :: l)
in next []
in
List.iter print_endline (List.rev wordlist)
```

8.1.9 scanner.mll

```
(* scanner.mll *)
(* @authors: Shuai Sun *)
(* modified by Jiabin Hu and Yan Zou on identifier and keyword order issues *)

{
open Parser

let rec convert_str ori tar =
  let convert_char c = match c with
    'n' -> '\n'
    | 'r' -> '\r'
    | 't' -> '\t'
    | 'b' -> '\b'
    | ’\’’ -> ’\’’
    | ’\"’ -> ’\"’
    | ’\\’ -> ’\\’
    | _ -> raise (Failure ("Illegal escpae character: "^"\\"^(String.make 1 c)))
in
  match ori with
    "" -> tar
    | _ -> if (String.get ori 0) = ’\\’ then convert_str (String.sub ori 2
        ((String.length ori) - 2)) (tar^(String.make 1 (convert_char (String.get ori
        1)))) else convert_str (String.sub ori 1 ((String.length ori) - 1))
        (tar^(String.make 1 (String.get ori 0)))
}

(* letter, digit *)
let letter = [’a’-’z’ ’A’-’Z’]
let digit = [’0’-’9’]
let character = [’\b’ ’\t’ ’\n’ ’\r’ ’\\’ ’ ’ ’!’ ’@’ ’#’ ’$’ ’%’ ’^’ ’&’ ’*’ ’(’ ’)’
    ’-’ ’_’ ’+’ ’=’ ’{’ ’[’ ’}’ ’]’ ’|’ ’;’ ’:’ ’<’ ’>’ ’.’ ’,’ ’?’ ’/’ ’~’ ’‘’]
            | letter | digit

rule token = parse
    | "/*"                 { comment lexbuf }
    | "//"                 { line_comment lexbuf }
    | [’ ’ ’\t’ ’\r’ ’\n’] { token lexbuf }

    (* Keywords *)
    | "if"            { IF }
    | "else"          { ELSE }
    | "while"         { WHILE }
    | "do"            { DO }
    | "for"           { FOR }
    | "break"         { BREAK }
    | "continue"      { CONTINUE }
    | "foreach"       { FOREACH }
    | "in"            { IN }
    | "by"            { BY }
    | "preorder"      { PREORDER }
    | "inorder"       { INORDER }
    | "postorder"      { POSTORDER }
    | "levelorder"    { LEVELORDER }
    | "return"        { RETURN }
    | "int"           { INT_T }
    | "float"         { FLOAT_T }
    | "char"          { CHAR_T }
    | "string"        { STRING_T }
    | "bool"          { BOOL_T }
    | "void"          { VOID }
    | "treetype"      { TREETYPE }
    | "main"          as main
                      { ID(main) }
    | "print"         as print
                      { ID(print) }
    | "alloc"         as alloc
                      { ID(alloc) }

    (* Constants *)
    | digit+          as integer
                      { INT(int_of_string integer) }
    | digit+ ’.’ digit*
    | ’.’ digit+ (’e’ [’+’’-’]? digit+)?
    | digit+ (’.’ digit+)? ’e’ [’+’ ’-’]? digit+      as float
                      { FLOAT(float_of_string float) }
    | "true"
    | "false"         as bool
                      { BOOL(bool_of_string bool) }
    | ’"’ (character | ’\’’)* ’"’ as string
                      { STRING(convert_str (String.sub string 1 ((String.length string)
                        - 2)) "") }
    | ’\’’ (character | ’’’ | ’"’) ’\’’ as string
                      { CHAR(String.get string 1) }
    | '\'' '\\' 'n' '\''
                      { CHAR('\n') }
    | '\'' '\\' 't' '\''
                      { CHAR('\t') }
    | '\'' '\\' 'b' '\''
                      { CHAR('\b') }
    | '\'' '\\' 'r' '\''
                      { CHAR('\r') }
    | '\'' '\\' '\\' '\''
                      { CHAR('\\') }

    | '~'             { NULL }


    | '{'    { LBRACE }
    | '}'    { RBRACE }
    | ';'    { SEMI }
    | ':'    { COLON }
    | ','    { COMMA }

    | '='    { ASSIGN }

    (* Operators *)
    | "->"   { CONNECT }
    (* These operators are removed.
    | "+="   { PLUS_ASN }
    | "-="   { MINUS_ASN }
    | "*="   { TIMES_ASN }
    | "/="   { DIVIDE_ASN }
    | "%="   { MOD_ASN }
    *)
    | "||"   { OR }
    | "&&"   { AND }
    | '!'    { NOT }
    | "!="   { NEQ }
    | '>'    { GT }
    | '<'    { LT }
    | "<="   { LEQ }
    | ">="   { GEQ }
    | "=="   { EQ }
    | '+'    { PLUS }
    | '-'    { MINUS }
    | '*'    { TIMES }
    | '/'    { DIVIDE }
    | '%'    { MOD }
    | '$'    { DOLLAR }
    | '@'    { AT }
    | '['    { LBRACK }
    | ']'    { RBRACK }
    | '&'    { DEG_AND }
    | '.'    { DOT }
    | '#'    { HASH }
    | '^'    { FATHER }
    | '('    { LPAREN }
    | ')'    { RPAREN }

  | letter (letter | digit | '_')* as identifier
           { ID(identifier) }

    | eof { EOF }
    | _ as err_char { raise (Failure("illegal character " ^ Char.escaped err_char)) }

(* comment *)
and line_comment = parse
    '\n'     { token lexbuf }
    | _      { line_comment lexbuf }

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

8.1.10 tml.ml

```
(* tml.ml *)
(* @authors: Yan Zou *)

if (Array.length Sys.argv < 2) then
  print_endline "Usage: tmlc <file_name>"
else
  let in_filename = Sys.argv.(1) in
  let out_filename =
    let slash_pos = (* the beginning of filename *)
      try String.rindex in_filename '/'
      with Not_found -> -1
    in
    let dot_pos = (* the end of filename *)
      try String.rindex in_filename '.'
      with Not_found -> String.length in_filename
    in
    let dot_pos = (* end should > begin, otherwise dot is for directory *)
      if dot_pos < slash_pos then
        (String.length in_filename)
      else
        dot_pos
    in
    (String.sub in_filename 0 dot_pos) ^ ".tmb"
  in
  let fin = open_in in_filename in
  let lexbuf = Lexing.from_channel fin in
  let program = Analyzer.check (Parser.program Scanner.token lexbuf) in
    ignore(close_in fin); Generator.translate out_filename program
```

8.1.11 type.mli

```
type t =
  | Int
  | Float
  | Char
  | String
  | Boolean
  | Void
  | Tree_type of string(* including return type of even main function *)

type literal =
    IntLit of int
  | FloatLit of float
  | CharLit of char
  | BoolLit of bool
  | StringLit of string
  | TreeLit (* can only be ~ *)

type op =
  | Add
  | Sub
  | Mult
  | Div
  | Mod
  | Equal
  | Neq
  | Less_than
  | Leq
  | Greater_than
  | Geq
  | Or
  | And
  | Not
  | Dollar
  | At
  | Deg_a
  | Hsh
  | Dot
  | Child
  | Father

type traverse_order =
  | Preorder
  | Inorder
  | Postorder
  | Levelorder (* different traversal orders *)
```

8.2 trunk/src/TML_Interpreter/

The following source code files are in alphabetical order.

8.2.1 InorderIterator.java

```java
import java.util.ArrayList;
import java.util.Stack;

/**
 * @author Jiabin Hu
 *
 */
public class InorderIterator implements TreeIterator
{
    private Stack<TMLTree> stack = new Stack<TMLTree>();

    public InorderIterator(TMLTree root)
    {
        TMLTree current = root;

        while (current != null)
        {
            stack.push(current);
            current = current.getChild(0);
        }
    }

    @Override
    public TMLTree next()
    {
        TMLTree result = stack.pop();

        TMLTree current = result.getChild(1);
        while (current != null)
        {
            stack.push(current);
            current = current.getChild(0);
        }

        return result;
    }

    @Override
    public boolean hasNext()
    {
        return !stack.isEmpty();
    }
}
```

8.2.2 Instruction.java

```java
/**
 * @author Jiabin Hu
 *
 */
public class Instruction
{
    public enum Type
    {
        Glb, Psi, Psf, Pss, Psb, Psc, Pop, Uop, Bin, Lod, Str, Lfp, Sfp, Jsr, Ent, Ret,
            Beq, Bne, Bra, Hlt,

        // Instruction of trees
        Alc, Fld, Sfd, Pst, Scd, Nxt
    }

    public enum SubType
    {
        // Binops
        Add, Sub, Mul, Div, Mod,

        Eq, Neq, Lt, Leq, Gt, Geq,

        And, Or,

        Val, Chd,

        // Uniops
        Neg, Not, Num, At, Cln, Fat
    }

    private Type type;
    private SubType subType;
    private Object operand;

    public Instruction(Type type, SubType subType, Object operand)
    {
        this.type = type;
        this.subType = subType;
        this.operand = operand;
    }

    public Type getType()
    {
        return type;
    }

    public SubType getSubType()
    {
        return subType;
    }

    public Object getOperand()
    {
        return operand;
    }
}
```

8.2.3 LevelorderIterator.java

```
import java.util.ArrayList;
import java.util.LinkedList;
import java.util.Queue;

/**
 * @author Jiabin Hu
 *
 */
public class LevelorderIterator implements TreeIterator
{
    Queue<TMLTree> queue = new LinkedList<TMLTree>();

    public LevelorderIterator(TMLTree root)
    {
        queue.add(root);
    }

    @Override
    public TMLTree next()
    {
        TMLTree result = queue.poll();

        ArrayList<TMLTree> children = result.getChildren();
        for (int i = 0; i < children.size(); ++i)
        {
            if (children.get(i) != null)
                queue.add(children.get(i));
        }

        return result;
    }

    @Override
    public boolean hasNext()
    {
        return !queue.isEmpty();
    }

}
```

8.2.4 Main.java

```
import java.io.BufferedReader;
import java.io.FileReader;
import java.util.ArrayList;
import java.util.Scanner;

/**
 * @author Jiabin Hu
 *
 */
public class Main
{
    public static void main(String[] args)
    {
        String filename = args[0];
        ArrayList<Instruction> instructions = new ArrayList<Instruction>();
        Scanner scanner = null;

        try
        {
            scanner = new Scanner(new BufferedReader(new FileReader(filename)));
            scanner.useDelimiter("\\s+");
            while (scanner.hasNext())
            {
                int lineNumber = scanner.nextInt();
                Instruction.Type type = null;
                Instruction.SubType subType = null;
                String instruction = scanner.next();
                try
                {
                    type = Instruction.Type.valueOf(instruction);
                } catch (IllegalArgumentException e)
                {
                    type = Instruction.Type.Bin;
                    subType = Instruction.SubType.valueOf(instruction);

                    switch (subType)
                    {
                    case Add:
                    case Sub:
                    case Mul:
                    case Div:
                    case Mod:
                    case Eq:
                    case Neq:
                    case Lt:
                    case Leq:
                    case Gt:
```

```
        case Geq:
        case And:
        case Or:
        case Val:
        case Chd:
            scanner.next();
            break;
        case Neg:
        case Not:
        case Fat:
        case Num:
        case At:
        case Cln:
            scanner.next();
            type = Instruction.Type.Uop;
            break;
        default:
            break;
        }
    }

    Object operand = null;
    switch (type)
    {
    case Glb:
    case Pop:
    case Psi:
    case Lod:
    case Str:
    case Lfp:
    case Sfp:
    case Jsr:
    case Ent:
    case Ret:
    case Beq:
    case Bne:
    case Bra:
    case Alc:
    case Sfd:
    case Nxt:
        operand = scanner.nextInt();
        break;
    case Psb:
        operand = scanner.nextBoolean();
        break;
    case Psc:
        operand = (char) scanner.nextInt();
        break;
    case Psf:
        operand = scanner.nextFloat();
        break;
    case Pss:
    {
        String line = scanner.nextLine();
        Scanner lineScanner = new Scanner(line);
        StringBuilder result = new StringBuilder();
        while (lineScanner.hasNextInt())
        {
            int ascii = lineScanner.nextInt();
            result.append((char) ascii);
        }
        operand = result.toString();
        break;
    }
    case Fld:
        operand = scanner.next().charAt(0);
        break;
    default:
        break;
    }
```

```
                instructions.add(new Instruction(type, subType, operand));
            }
        } catch (Exception e)
        {
            System.err.println("Error␣when␣processing␣input␣file.");
            e.printStackTrace();
            System.exit(1);
        } finally
        {
            if (scanner != null)
                scanner.close();
        }

        new Program(0, instructions).execute();
    }
}
```

8.2.5 PostorderIterator.java

```
import java.util.LinkedList;
import java.util.Queue;

/**
 * @author Jiabin Hu
 *
 */
public class PostorderIterator implements TreeIterator
{
    Queue<TMLTree> queue = new LinkedList<TMLTree>();

    public PostorderIterator(TMLTree root)
    {
        buildQueue(root);
    }

    private void buildQueue(TMLTree node)
    {
        for (TMLTree child : node.getChildren())
        {
            if (child != null)
                buildQueue(child);
        }

        queue.add(node);
    }
    @Override
    public TMLTree next()
    {
        return queue.poll();
    }

    @Override
    public boolean hasNext()
    {
        return !queue.isEmpty();
    }

}
```

8.2.6 PreorderIterator.java

```
import java.util.Stack;
import java.util.ArrayList;

/**
 * @author Jiabin Hu
 *
 */
public class PreorderIterator implements TreeIterator
```

```
{
    private Stack<TMLTree> stack = new Stack<TMLTree>();

    public PreorderIterator(TMLTree root)
    {
        stack.push(root);
    }

    @Override
    public TMLTree next()
    {

        TMLTree result = stack.pop();

        ArrayList<TMLTree> children = result.getChildren();
        for (int i = children.size() - 1; i >= 0; --i)
        {
            if (children.get(i) != null)
                stack.push(children.get(i));
        }
        return result;

    }

    @Override
    public boolean hasNext()
    {
        return !stack.isEmpty();
    }

}
```

8.2.7 TMLTree.java

```
import java.util.ArrayList;

/**
 * @author Jiabin Hu
 *
 */
public class TMLTree
{
    static private int count = 0;
    private TMLTree parent;
    private ArrayList<TMLTree> children;
    private ArrayList<Object> data;

    private int id;

    public int getId()
    {
        return id;
    }

    public TMLTree(int numOfChildren)
    {
        this(numOfChildren, null);
    }

    public TMLTree(int numOfChildren, TMLTree parent)
    {
        data = new ArrayList<Object>();
        children = new ArrayList<TMLTree>(numOfChildren);
        for (int i = 0; i < numOfChildren; i++)
            children.add(null);
        this.parent = parent;
        this.id = TMLTree.count++;
    }

    public void addData(Object data)
    {
        this.data.add(data);
    }

    @SuppressWarnings("unchecked")
    public TMLTree cloneNode()
    {
        TMLTree result = new TMLTree(this.children.size());
        result.data = (ArrayList<Object>) this.data.clone();
        return result;
    }

    @SuppressWarnings("unchecked")
    public TMLTree cloneTree(boolean isRoot)
    {
        int degree = this.children.size();
        TMLTree result = new TMLTree(degree);
        result.data = (ArrayList<Object>) this.data.clone();
        result.parent = isRoot ? null : this.parent;
        for (int i = 0; i < degree; i++)
        {
            if (children.get(i) == null)
                result.children.set(i, null);
            else
                result.children.set(i, children.get(i).cloneTree(false));
        }

        return result;
    }

    public int findChild(TMLTree child)
    {
        int result = -1;
        for (int i = 0; i < children.size(); i++)
            if (children.get(i) == child)
                result = i;

        return result;
    }

    public TMLTree getChild(int index)
    {
        return children.get(index);
    }

    public ArrayList<TMLTree> getChildren()
    {
        return children;
    }

    public Object getData(int index)
    {
        return data.get(index);
    }

    public TMLTree getParent()
    {
        return parent;
    }

    public void setChild(int index, TMLTree child)
    {
        this.children.set(index, child);
    }

    public void setData(int index, Object data)
    {
        this.data.set(index, data);
    }

    public void setParent(TMLTree parent)
    {
        this.parent = parent;
    }

    @Override
    public String toString()
    {
        StringBuilder result = new StringBuilder();
        result.append("Tree␣" + id + ":␣[p="
                + ((parent == null) ? -1 : parent.id) + ",␣c=(");
        if (children != null)
        {
            for (TMLTree child : children)
                if (child != null)
                    result.append(child.id + ",");
        }
        result.append("),␣d=(");
        if (data != null)
        {
            for (Object d : data)
                result.append(d.toString() + ",");
        }
        result.append(")]");

        return result.toString();
    }

}
```

8.2.8 TreeIterator.java

```
/**
 * @author Jiabin Hu
 *
 */
public interface TreeIterator
{
    TMLTree next();

    boolean hasNext();
}
```

8.3 truck/test/SrcCode

The following test case source files are in alphabetical order.

8.3.1 arithmetic_op.tml

```
void main()
{
    print(1 + 2);
    print(-1 + 5);
    print(3 * 4);
    print(3 - 4);
    print(4 / 2);
    print(5 / 3);
    print(5 % 3);
    print(1 + 2 * 3 / 4 % 3 - 7);
}
```

8.3.2 assignment_op.tml

```
void main()
{
    int i;
    i = 1;
    print(i);
    char j;
    j = 'a';
    print(j);

    float k;
    k = 0.01;
    print(k);

    string s;
    s = "hi";
    print(s);
}
```

8.3.3 comparison_op.tml

```
void main()
{
    print(1 < 2);
    print(1 > 2);
    print(2 >= 2);
    print(2 <= 3);
    print(1 != 3);
    print(3 == 4);
}
```

8.3.4 complex1.tml

```
int y=10;
int foo()
{
    int k=10;
    y = y +1;
    return k;
}
int main()
{
    int i;
    int j = foo();
    print("Does that work...?");
    for(i=0;i<10;i=i+1)
    {
        if(i == 6)
        {
            print("I am getting outaa here..");
            break;
        }
        print("Currently i is..",i);
    }
    print("J is..", j);
}
```

8.3.5 complex2.tml

```
int i;
treetype <6, [a,b,c,d,e,f]> MyTree_t
{
    int num;
    float numf;
    string nums;
    char numc;
}

treetype <3> NewTree
{
    int a;
    float b;
    string c;
}
```

```
void main ()
{
    MyTree_t ta, tb,tc,td,te,tf;
    NewTree a1,a2;

    alloc(ta, tb, tc, td, te, tf);
    alloc(a1,a2);

    int childnum1,childnum2;

    ta ->(tb:~:(tf->(tc:td->(~:te))):~:~:~);
    a1->(a2:~);
    childnum1 = #tb;
    childnum2 = #a2;
    te = $tf;
    td = @tf;
    te = tf;
    ta[0].num = 1;

    if(childnum1 == childnum2)
    {
        print("position of both nodes is same");
    }
    int j;
    for(j=0;j<childnum1+1;j=j+1)
    {
        print(ta[j].num);
        ta[i].nums="hello world";
    }
}
```

8.3.6 compound.tml

```
void main()
{
    {
        {
            {
                {
                    {
                        {
                            {
                                {
                                }
                            }
                        }
                    }
                }
            }
        }
    }
}
```

8.3.7 continue.tml

```
int main()
{
    int i=10;
    while(i>0)
    {
        i=i-1;
        if(i>6)
              continue;
        print("hello world");
    }
}
```

8.3.8 dangling_else.tml

```
void main()
{
    if (true) if (false) print(1); else print(2);
    if (false) if (true) print(1); else print(2);
    if (false) if (false) print(1); else print(2);
    if (true) if (true) print(1); else print(2);
}
```

8.3.9 do_while.tml

```
void main()
{
    int i = 0;
    do
    {
        print(i);
        i = i + 1;
    }while(i < 5);
}
```

8.3.10 empty_str.tml

```
int main()
{
    print("");
}
```

8.3.11 empty.tml

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

8.3.12 for.tml

```
void main()
{
    int i;
    for (i = 0; i < 10; i = i + 1)  print(i);
    for (i = 10; i > 0; i = i - 2)
    {
        print('\n');
        print(i);
    }
}
```

8.3.13 foreach_3nodes.tml

```
treetype <3> MyTree_t
{
    int val = 0;
}

void main ()
{
    int i = 0;
    MyTree_t a, b, c, d, e, f;
    alloc(a, b, c, d, e, f);

    a -> (b : c -> (e : f : ~) : d);

    foreach node in a by preorder
    {
        node.val = i;
        i = i + 1;
    }

    foreach node in a by preorder
    {
        print (node.val);
        print (" ");
    }
    print ('\n');

    foreach node in a by postorder
    {
        print (node.val);
        print (" ");
    }
    print ('\n');

    foreach node in a by levelorder
    {
        print (node.val);
        print (" ");
    }
}
```

8.3.14 foreach_single.tml

```
treetype <2> MyTree_t
{
    int val = 0;
}


void main ()
{
    int i = 0;
    MyTree_t a;
    alloc(a);

    foreach node in a by preorder
    {
        node.val = i;
        i = i + 1;
    }

    foreach node in a by preorder
    {
        print (node.val);
        print (" ");
    }
    print ('\n');

    foreach node in a by inorder
    {
        print (node.val);
        print (" ");
    }
    print ('\n');

    foreach node in a by postorder
    {
        print (node.val);
        print (" ");
    }
    print ('\n');

    foreach node in a by levelorder
    {
        print (node.val);
        print (" ");
    }
}
```

8.3.15 foreach.tml

```
treetype <2> MyTree_t
{
    int val = 0;
}


void main ()
{
    int i = 0;
    MyTree_t a, b, c, d, e, f;
    alloc(a, b, c, d, e, f);

    a -> (b -> (d : e -> (f : ~)): c);

    foreach node in a by preorder
    {
        node.val = i;
        i = i + 1;
    }

    foreach node in a by preorder
    {
        print (node.val);
        print (" ");
    }
    print ('\n');

    foreach node in a by inorder
    {
        print (node.val);
        print (" ");
    }
    print ('\n');

    foreach node in a by postorder
    {
        print (node.val);
        print (" ");
    }
    print ('\n');

    foreach node in a by levelorder
    {
        print (node.val);
        print (" ");
    }
}
```

8.3.16 function_forloop.tml

```
int foo()
{
    int k=10;
    return k;
}
int main()
{
    int i;
    int j = foo();
    print("Does that work...?");
    for(i=0;i<3;i=i+1)
    {
        print("Currently i is..",i);
    }
    print("J is..", j);
}
```

8.3.17 function.tml

```
int foo()
{
    print("This is a function foo\n");
    return 2;
}

void main()
{
    int a = foo();
    print("Function foo returns ", a, '\n');
}
```

8.3.18 gcd.tml

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

void main()
{
    print(gcd(2,14), '\n');
    print(gcd(3,15), '\n');
    print(gcd(99,121), '\n');
}
```

8.3.19 global_var.tml

```
int a;
char b;
string c;
float d;

int main()
{
}
```

8.3.20 if.tml

```
void main()
{
    if (2 > 1)
    {
        print("2 > 1\n");
    }
    else
    {
        print("2 < 1\n");
    }
    if (1 < 2) print("1 < 2\n");
}
```

8.3.21 logical_op.tml

```
void main()
{
    print(true && true);
    print(false && true);
    print(true && false);
    print(false && false);
    print('\n');

    print(true || true);
    print(false || true);
    print(true || false);
    print(false || false);
    print('\n');

    print(!true);
    print(!false);
}
```

8.3.22 loop_control.tml

```
void main()
{
    int i;
    for (i = 0; true; i = i + 1)
    {
        if (i % 2 == 0) continue;
        if (i > 5) break;
        print(i);
    }
}
```

8.3.23 main.tml

```
void main()
{
}
```

8.3.24 print_backslash_chars.tml

```
void main()
{
    print("the \b: ");
    print('\b');
    print("the\n:");
    print('\n');
    print("the \r: ");
    print('\r');
    print("the \t: ");
    print('\t');
    print("the \\: ");
    print('\\');
}
```

8.3.25 print_char.tml

```
void main()
{
    print('1');
    print('a');
    print('(');
}
```

8.3.26 print_string.tml

```
void main()
{
    print("abcde");
}
```

8.3.27 recursion.tml

```
int fib(int n)
{
    if (n < 1) return 0;
    if (n == 1) return 1;
    return (fib(n - 1) + fib(n - 2));
}

void main()
{
    print(fib(10));
}
```

8.3.28 scope.tml

```
int a = 1;

void fun1()
{
    print(a);
}

void fun2()
{
    int a = 2;
    print(a);
}

void main()
{
    fun1();
    fun2();
}
```

8.3.29 tree_type1.tml

```
treetype <2> binary_tree
{
    int a;
    float b;
}

int main()
{
}
```

8.3.30 tree_type2.tml

```
treetype <2, [left, right]> binary_tree
{
    int a;
    float b;
}
int main()
{
}
```

8.3.31 tree_type3.tml

```
treetype <1> list
{
    int k2;
    int v;
}

treetype <1, [next]> table
{
    int key1;
    list l;
}

int main()
{
}
```

8.3.32 tt_connect.tml

```
treetype <2, [left, right]>MyTree_t
{
    int vint = 0;
    float vflt = 1.;
    string vstr = "hi";
}


void main ()
{
    MyTree_t ta, tb, tc;
    alloc(ta, tb, tc);

    ta.vstr = "root";
    tb.vstr = "left child";
    tc.vstr = "right child";

    ta ->(tb: tc);
    print (ta.vstr);         // root
    print (ta[left].vstr);  // left child
    print (ta[1].vstr);      // right child

}
```

8.3.33 tt_connect2.tml

```
treetype <2, [left, right]>MyTree_t
{
    int vint = 0;
    float vflt = 1.;
    string vstr = "hi";
}


void main ()
{
    MyTree_t ta, tb, tc;
    alloc(ta, tb, tc);

    ta.vstr = "root";
    tb.vstr = "i'm tb";
    tc.vstr = "i'm tc";


    ta -> ( ~ : tb -> ( ~ :tc));


    print (ta.vstr);                 // root
    print (ta[1].vstr);              // i'm tb
    print (ta[1][right].vstr);       // i'm tc

}
```

8.3.34 tt_data.tml

```
treetype<1> Data
{
    int a;
    //char b;
    //float c;
}
```

```
int main()
{
    Data node;
    alloc(node);
    node.a = 1;
    //node.b = 'a';
    //node.c = 2.0;
    print(node.a);
    //print(node.b);
    //print(node.c);
    print('\n');
}
```

8.3.35 tt_degree.tml

```
treetype <2, [left, right]>MyTree_t
{
    int vint;
    float vflt;
    string vstr;
}


void main ()
{
    MyTree_t ta, tb, tc;
    alloc(ta, tb, tc);
    int dgr1, dgr2, dgr3;

    ta ->(tb: tc);

    dgr1 = &ta;
    dgr2 = &tb;
    dgr3 = &tc;

    //print (dgr1 == dgr2 == dgr3);     // true
    print (dgr1 == dgr2);
    print (dgr1 == dgr3);
    print('\n');
}
```

8.3.36 tt_hash.tml

```
treetype <2, [left, right]>MyTree_t
{
    int vint;
    float vflt;
    string vstr;
}


void main ()
{
    MyTree_t ta, tb, tc;
    alloc(ta, tb, tc);
    int ta_r, tb_r, tc_r;

    ta ->(tb: tc);

    ta_r = #ta;
    tb_r = #tb;
    tc_r = #tc;

    print (ta_r);       // -1
    print (tb_r);       // 0
    print (tc_r);       // 1

}
```

8.3.37 tt_init_value_nest.tml

```
treetype <1> list
{
    int k2 = -1;
    int v;
}

treetype <1, [next]> table
{
    int key1 = 1;
    list l;
}

void main()
{
    table t1;
    alloc (t1);


    print (t1.key1);     // 1
    print (t1.l == ~);  // true

    alloc (t1.l);
    print (t1.l.k2);     // -1
    print (t1.l.v);      // 0
}
```

8.3.38 tt_init_value.tml

```
treetype <2, [left, right]> Bin_T
{
    int i = 1000;
    int i2;
    float f = 1e-2;
    string s = "hello, TML.";
}

void main()
{
    Bin_T t1;
    alloc (t1);

    print (t1.i);
    print (t1.f);
    print (t1.s);
    print (t1.i2);  //0
}
```

8.3.39 tt_node_copy.tml

```
treetype <2, [left, right]>MyTree_t
{
    int vint = 0;
    float vflt = 1.;
    string vstr = "hi";
}


void main ()
{
    MyTree_t ta, tb, tc;
    alloc(ta, tb, tc);

    ta.vstr = "root";
    tb.vstr = "left child";
    tc.vstr = "right child";

    ta ->(tb: tc);
    MyTree_t ta_c1, ta_c2;

    ta_c1 = @ta;

    if (ta_c1[left] != ~ || ta_c1[1] != ~ )
    {
        print ("error: should be null");
        print ('\n');
    }

    print (ta_c1.vstr);     // root


}
```

8.3.40 tt_parent.tml

```
treetype <2, [left, right]>MyTree_t
{
    int vint = 0;
    float vflt = 1.;
    string vstr = "hi";
}

void main ()
{
    MyTree_t ta, tb, tc;
    alloc(ta, tb, tc);

    ta ->(tb: tc);


    print (ta == ^tb);      // true
    print (ta == ^tc);          // true
    print (^ta == ~);           // true

}
```

8.3.41 tt_tr_copy.tml

```
treetype <2, [left, right]>MyTree_t
{
    int vint;
    float vflt;
    string vstr;
}


void main ()
{
    MyTree_t ta, tb, tc;
    alloc(ta, tb, tc);

    ta.vstr = "root";
    tb.vstr = "left child";
    tc.vstr = "right child";

    ta ->(tb: tc);

    MyTree_t ta_c2;

    ta_c2 = $ta;

    if (ta_c2[left] != ~ && ta_c2[1] != ~ )
    {
        print (ta_c2[left].vstr);   // left child
        print (ta_c2[1].vstr);      // right child
    }
}
```

8.3.42 var_init.tml

```
void main()
{
    int a = 0;
    char b = 'b', b2 = '2', b3 = '3';
    string c = "Hello world";
    float d = 1.11;
    bool e = true;
    print(a, b, b2, b3, c, d, e);
}
```

8.3.43 while.tml

```
void main()
{
    int i = 5;
    while (i < 10)
    {
        print(i, '\n');
        i = i + 1;
    }
}
```