TML
Language Reference Manual

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

Columbia University

October 31, 2011

Contents

1 Introduction

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.

2 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.

2.1 Character Set

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

2.2 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.
```

2.3 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.

2.4 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.

2.5 Constants

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

2.5.1 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.

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

2.5.2 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.

2.5.3 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.

2.5.4 String constants

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

2.5.5 Boolean constants

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

2.5.6 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 Types

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

3.1 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 5.1.1

3.1.1 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.

3.1.2 Floating point numbers

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

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

Table 2: Primitive Types in TML

3.1.3 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.

3.1.4 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.

3.1.5 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 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.

3.2.1 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 defiined. The size of a tree type is the sum size of its member variable types.

When defining each member variable, an intial value can be optimally defined. If the initial value of a member variable is not defined, it will be intialized 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 ";
    }
```

3.2.2 Type name

The *type name* of a tree type follows the identifier definition in the section 2.3. After the tree type is defined, the type name can be used to define variables of this type.

3.2.3 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 4.6.3.

3.2.4 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 4.6.3.

3.2.5 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 2.3. 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 4.6.3.

3.3 Type Conversions

In TML, no type conversion is allowed.

4 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.

4.1 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.

4.2 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.

4.3 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.

4.4 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.

4.5 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.

4.6 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.

4.6.1 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 seperated 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);
```

4.6.2 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
```

4.6.3 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 3.2.1
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
```

4.7 Precedence Rules

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

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

Table 3: Precedence Rules

5 Statements

5.1 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 → *initializer* | *initializer-list* , *initializer*

5.1.1 Type Specifiers

Type Specifiers can be any basic type or user-defined tree type.
For basic types, it can be:

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

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

5.1.2 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 → identifier [= *expression*]

5.2 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;

5.3 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 → *statement* | *statement statement-list*

5.4 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);
}
```

5.5 Iterative Statements

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

5.5.1 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*

5.5.2 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*);

5.5.3 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-expresion 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 afterwards. It has the following syntax:

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

5.5.4 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.

5.6 Other Statements

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

5.6.1 break statements

This statement is simply written as:

```
break;
```

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

5.6.2 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.

5.6.3 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.

5.6.4 Empty statements

This statement is simply written as:

```
;
```

It does nothing.

6 Functions

6.1 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 5.1.1. 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);
}
```

6.2 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
}
```

6.3 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.

6.3.1 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.

6.3.2 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.

7 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.

8 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.

9 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 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 }
```

And the output of the parser in file "parser.output" is:

72 terminals, 22 nonterminals

91 grammar rules, 190 states