LaTenS Language Reference Manual

Authors:
Mohit Rajpal
Daniel Schwartz
Elsbeth Turcan
Eliana Ward-Lev

October 26, 2016

Contents

1 Introduction

1.1 Motivation and Introduction to Tensors

"Pure mathematics is, in its way, the poetry of logical ideas." - Albert Einstein

"Tensors are mathematical objects that can be used to represent real-world systems. [...] Tensors have proven to be useful in many engineering contexts, in fluid dynamics, and for the General Theory of Relativity. Knowledge of tensor math [...] also is useful in the fields of financial analysis, machine [learning], and in the analysis of other complex systems." [1]

Tensors are essentially the generalization of a matrix or vector to arbitrary rank. Rank refers to the number of indices needed to label a component of that tensor. For example, a scalar is a rank zero tensor, a vector is a rank one tensor and a matrix is a rank two tensor.
Indices are used to denote a particular element of a tensor. For example, the formula to find the eigenvectors of a matrix M is

$$\begin{pmatrix} M_{00} & M_{01} & M_{02} \\ M_{10} & M_{11} & M_{12} \\ M_{20} & M_{21} & M_{22} \end{pmatrix} \begin{pmatrix} v_0 \\ v_1 \\ v_2 \end{pmatrix} = \lambda \begin{pmatrix} v_0 \\ v_1 \\ v_2 \end{pmatrix}$$

Using index notation, the eigenvector formula becomes

$$\sum_{j=0}^{3} M_{ij} v_j = \lambda v_i$$

Einstein summation convention allows for dropping of the sign

$$M_{ij} v_j = \lambda v_i$$

Note that Einstein summation follows a simple rule: any pair of indices with the same symbol implies summation. Also note that any index without a pair must also apear on the other side of the equality (here i).

1.2 Language Description

LaTenS is a language that strives to make calculations with tensors as simple as possible. We follow and extend the customs of MATLAB, generalizing a matrix-manipulation language to manipulate n-dimensional tensors. The syntax is intended to evoke traditional mathematical syntax for operating on tensors—for example, when composing in LaTeX, a tensor $T_{a,b}$ can be created by typing `T_{a,b}`, so this syntax is likely familiar to mathematicians.

LaTenS uses this kind of syntax to make problems involving tensor manipulation quick and intuitive, with built-in operations on n-dimensional tensors without the need to loop over elements in the tensors (as in other programming languages).

2 Lexical Conventions

2.1 Tokens

A LaTenS program is parsed into tokens according to the following scanner. Most important are the operators, and then the reserved keywords. Comments and whitespace are ignored (except to separate tokens).

```
[' ' '\t' '\r' '\n'] { token lexbuf } (* Whitespace *)
| "%{"  { comment lexbuf } (* Block comments *)
| '%'  { singlecomment lexbuf } (* Single-line comments *)
| '('  { LPAREN }
```

```
| ')'  { RPAREN }
| '['  { LBRACKET }
| ']'  { RBRACKET }
| '{'  { LBRACE }
| '}'  { RBRACE }
| ';'  { SEMI }
| ','  { COMMA }
| '_'  { UNDER }
| '+'  { PLUS }
| '-'  { MINUS }
| '*'  { TIMES }
| '/'  { DIVIDE }
| '^'  { CONCAT }
| "==" { EQ }
| '='  { ASSIGN }
| "~=" { NEQ }
| '<'  { LT }
| "<=" { LEQ }
| '>'  { GT }
| ">=" { GEQ }
| "&&" { AND }
| "||" { OR }
| '~'  { NOT }
| "else"      { ELSE }
| "float"     { FLOAT }
| "for"       { FOR }
| "if"        { IF }
| "let"       { LET }
| "return"    { RETURN }
| "string"    { STRING }
| "tensor"    { TENSOR }
| "while"     { WHILE }
```

2.2 Comments

LaTenS supports single line comments

```
% Single Line Comment
```

and block comments over multiple lines

```
%{ Block Comment %}
```

Comments may not be nested, although the two types may be combined.

```
%{
    % This comment structure is also valid,
    % Although really everything inside a block comment is ignored,
    % So the single-line comments are not actually being recognized.
%}
```

2.3 Identifiers

An identifier name must be a letter (uppercase or lowercase) followed by any number of uppercase or lowercase letters and digits. No other characters may be included in an identifier; note that in particular,

programmers may expect the underscore (_) to be a legal identifier character, but see Section 3.2 (Tensor) for an explanation of why the underscore character is reserved.

2.4 Key Words

No identifier name can be a keyword reserved by the system. The following words are used elsewhere (control flow, for example) and are reserved:

- else
- float
- for
- if
- let
- return
- string
- tensor
- while

Reserved words are case-sensitive, so for example "If" is a valid identifier name.

3 Data Types

```
typ: FLOAT { Float }
   | STRING { String }
   | VOID { Void }
   | ID UNDER LBRACE qual_list { TensorType($4) }
```

```
var_typ: FLOAT { Float }
  | STRING { String }
  | ID UNDER LBRACE qual_list { TensorType($4) }
```

LaTenS is statically typed with hybrid type-checking, following the conventions of Swift or GO. This implies that variable types are inferred, but function return value must specified.

Every variable has a type (rank n tensor or string) and can be used in assignment as long as the right-hand side is of the same type (that is, typing is not dynamic; if a variable already has a type, it cannot be changed by assignment). In particular, this means that when assigning to a tensor, the expression on the right-hand side must have the same rank as the tensor variable, although the size of each dimension and which indices are up and down may vary.

3.1 Declaration

```
vdecl: LET ID UNDER LBRACE qual_list SEMI { TensorBind($2,$5)}
   | LET ID UNDER LBRACE qual_list ASSIGN expr SEMI { TensorBindAndAssign($2, $5, $7) }
   | LET ID ASSIGN expr SEMI { InferredBind($2, $4) }
   | LET ID SEMI { ($2) }
```

To declare a variable, use the keyword `let`. A variable can be declared only, or declared and assigned a value at the same time. The `let` keyword is used to differentiate declaration with assignment from just assignment, since without `let`, assignment would look the same with or without declaring a new variable (which could lead to hard-to-catch bugs). To illustrate declaration,

```
let T_{a,b,c};
let b = 10.5;
let myT_{a} = (3,)
```

are all valid declarations: the first declares a rank three tensor `T` (with no set values or dimensions), the second declares a floating-point number(scalar) and gives it the value `10.5`, and the third declares a rank 1 tensor of size three (this implicitly sets all values to 0). The declarations for each data type and details on float and string literals are given in more detail below.

Only one variable can be declared per declaration; that is, each new variable must have its own `let` keyword. Variables which have not been initialized cannot be used.

3.2 Tensor

```
tensorimmediate: LBRACKET ten_list RBRACKET { TenDim($2) }
| LBRACKET ten_expr_list RBRACKET { ExprDim($2) }
```

Tensors can have any rank. To declare a tensor, an identifier name must be used followed by an underscore (_) and in braces dummy indices separated by commas. The total number of dummy indices represents the rank. If there are no indices, it is a rank 0 tensor or a scalar and will be interpreted as a float(see 3.3).

```
qual_list: SLITERAL RBRACE { [$1] }
  | SLITERAL COMMA qual_list { $1 :: $3 }

 qual_list_expr: LPAREN expr RPAREN RBRACE { [$2] }
  | FLITERAL RBRACE { Float($1) }
  | SLITERAL RBRACE { String($1) }
  | LPAREN expr RPAREN COMMA qual_list_expr { $2::$5 }
  | FLITERAL COMMA qual_list_expr { $1::$3 }
  | SLITERAL COMMA qual_list_expr { $1::$3 }
```

To define a tensor, one can use nested square brackets, with the number of nested brackets being the same as the rank. Each element of the tensor must either be a float or a pre-initialized scalar. Additionally, if one desires to initialize a tensor with all 0's, one can use `( dim1, dim2, ..., dim n)`. If the rank is 1 use a comma after the dimension i.e. `(dim1,)`
For example, to declare a row vector (rank 1 tensor) $v_i = (1, 2, 3)$, and a matrix(rank 2 tensor) $M = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}$

```
let v_{i} = [1, 2, 3];
let M_{i,j} = (3, 3);
```

A tensor's type includes the rank but not whether indices are up or down or the size of each dimension.

3.3 Float

The floating-point type represents rational numbers. Any number that can be expressed as a terminating decimal can be expressed as a float in LaTenS.

A floating-point literal consists of an integer part, a decimal point, a fractional part, and an exponent.

Either the integer part or the fractional part (along with the decimal point) may be omitted (but not both), and the exponent is optional and may be optionally signed. This is the same definition of a floating-point number used by C.[2] All floats are signed; the unary minus (−) must be included to signify a negative number while the unary plus (+) is optional for convenience.

The following are some examples of assigned floats:

```
let a = 2;
let b = 1.7;
let c = -0.8
let d = .2
let e = +3.7
let f = 5.2e8
let g = 2e-5
```

Additionally, every element inside a tensor is also a floating-point number. A scalar is also a floating-point number.

3.4 String Literals

Similar to C, a string literal is a sequence of zero or more characters, digits, and escapable characters enclosed within double quotation marks.[3] As our language is geared towards scientific computation involving tensors, strings are designed to be used solely for debugging. For this reason, certain operations (such as retrieving the length of a string) are not allowed. Rather, strings are to be constructed, concatenated, and printed as necessary.

The following are valid escapable characters:

- newline: "\n"
- tab: "\t"
- backslash: "\\"
- single quote "\'"
- double quote: "\""

Strings can be concatenated with the ^ operator. (See section 4.2)

To express a string literal, use double quotes:

```
let str = "Hello";
str ^ " world!";
```

will return "Hello world"

3.5 Void

The void type is unique: no void variable can exist, and no operators operate on a void type, but a function may have void as a return type. This is simply for the convenience of defining a function that does not return any value.

4 Expressions and Operators

There are three possible types of expressions - regular expressions (which can be found in statements and so on); tensor expressions (which can be used inside a tensor immediate–so for example strings are not allowed);

and index expressions (which can be used inside tensor indices–so for example tensor immediates are not allowed).

```
expr:
    SLITERAL                    { String($1) }
  | FLITERAL        { Float($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 CONCAT expr   { Binop($1, Concat, $3) }
  | expr EQ expr               { Binop($1, Equal, $3) }
  | expr NEQ expr            { Binop($1, Neq, $3) }
  | expr LT expr               { Binop($1, Less, $3) }
  | expr LEQ expr            { Binop($1, Leq, $3) }
  | expr GT expr               { Binop($1, Greater, $3) }
  | expr GEQ expr            { Binop($1, Geq, $3) }
  | expr AND expr            { Binop($1, And, $3) }
  | expr OR expr               { Binop($1, Or, $3) }
  | MINUS expr %prec NEG  { Unop(Neg, $2) }
  | PLUS expr       { $2 (* Unary plus is redundant; ignore it *) }
  | NOT expr                   { Unop(Not, $2) }
  | ID             { Id($1) }
  | ID ASSIGN expr           { Assign($1, $3) }
  | LPAREN expr RPAREN  { $2 }
  | ID LPAREN actuals_opt RPAREN { Call($1, $3) }
  | ID UNDER LBRACE qual_list_expr { TensorId($1, $4) }
  | ID UNDER LBRACE qual_list ASSIGN expr { TensorAssign($1, $4, $6) }
  | tensorimmediate { Tensor($1) }
  | vdecl  { $1 }

ten_expr:
  FLITERAL         { Float($1) }
  | ten_expr PLUS ten_expr    { Binop($1, Add, $3) }
  | ten_expr MINUS ten_expr   { Binop($1, Sub, $3) }
  | ten_expr TIMES ten_expr   { Binop($1, Mult, $3) }
  | ten_expr DIVIDE ten_expr  { Binop($1, Div, $3) }
  | ten_expr EQ ten_expr   { Binop($1, Eq, $3) }
  | ten_expr NEQ ten_expr  { Binop($1, Neq, $3) }
  | ten_expr LT ten_expr   { Binop($1, Less, $3) }
  | ten_expr LEQ ten_expr  { Binop($1, Leq, $3) }
  | ten_expr GT ten_expr   { Binop($1, Greater, $3) }
  | ten_expr GEQ ten_expr  { Binop($1, Geq, $3) }
  | ten_expr AND ten_expr  { Binop($1, And, $3) }
  | ten_expr OR ten_expr   { Binop($1, Or, $3) }
  | MINUS ten_expr %prec NEG  { Unop(Neg, $2) }
  | PLUS ten_expr     { $2 }
  | NOT ten_expr      { Unop(Not, $2) }
  | ID             { Id($1) }
  | ID ASSIGN ten_expr     { Assign($1, $3) }
  | LPAREN ten_expr RPAREN  { $2 }
  | ID LPAREN actuals_opt RPAREN { Call($1, $3) }
  | LET ID ASSIGN ten_expr  { InferredBind($2, $4) }

ind_expr:
  FLITERAL         { Float($1) }
  | SLITERAL          { String($1) }
  | ind_expr PLUS ind_expr  { Binop($1, Add, $3) }
  | ind_expr MINUS ind_expr   { Binop($1, Sub, $3) }
```

```
| ind_expr TIMES ind_expr   { Binop($1, Mult, $3) }
| ind_expr DIVIDE ind_expr  { Binop($1, Div, $3) }
| ind_expr CONCAT ind_expr  { Binop($1, Concat, $3) }
| ind_expr EQ ind_expr   { Binop($1, Eq, $3) }
| ind_expr NEQ ind_expr  { Binop($1, Neq, $3) }
| ind_expr LT ind_expr   { Binop($1, Less, $3) }
| ind_expr LEQ ind_expr  { Binop($1, Leq, $3) }
| ind_expr GT ind_expr   { Binop($1, Greater, $3) }
| ind_expr GEQ ind_expr  { Binop($1, Geq, $3) }
| ind_expr AND ind_expr  { Binop($1, And, $3) }
| ind_expr OR ind_expr   { Binop($1, Or, $3) }
| MINUS ind_expr %prec NEG  { Unop(Neg, $2) }
| PLUS ind_expr     { $2 }
| NOT ind_expr      { Unop(Not, $2) }
| ID             { Id($1) }
| ID ASSIGN ind_expr     { Assign($1, $3) }
| LPAREN ind_expr RPAREN  { $2 }
| ID LPAREN actuals_opt RPAREN { Call($1, $3) }
| LET ID ASSIGN ind_expr  { InferredBind($2, $4) }
| ID UNDER LBRACE qual_list_expr { TensorId($1, $4) }
```

4.1 Unary Operators

Operator	Description	Example
`inv(Tensor)`	invert a tensor	`let T_a,b = [[1, 2], [3, 4]]; inv(T_a,b)` returns `[[−2, 1],[1.5, −.5]]`
`rank(Tensor)`	get rank of tensor	`let T = 5; rank(T);` returns 0
`dim(Tensor, Index)`	get size of a particular dimension	`let T_a,b = [[1, 2, 3], [3, 4, 5]]; dim(T_m,n, m)` returns 3.
`_ (underscore)`	get element by indices(to index using a predefined scalar, use ())	`let T_a,b = [[1, 2, 3], [3, 4, 5]]; a = 2; T_0,0;` returns 1 `T_(a),0;` returns 5
`+, −`	optional sign. − negates all indices. + is just cosmetic	`let T_a = [1, 2, 3]; −T_a;` returns `[−1, −2, −3]`
`str()`	return tensors as string	`str(1), let T_a,b = [[1, 2, 3], [3, 4, 5]]; str(T_a^bc)` returns `"1" "[[1, 2, 3], [3, 4, 5]]"`
`~`	negate the truth value of an expression	~0 is true and ~1 is false

4.2 Binary Arithmetic Operators

For a binary operation to be performed on two tensors, it is required that [[TODO ??? put in for each op]]

- `+`
  Tensor addition. Examples: `T_a+T_a, T_{a} + T_{b}`
  Both tensors must be the same rank if sizes differ we will 0 pad.

- `−`
  Tensor subtraction. Examples: `T_{a}−T_{a}, T_{a} − T_{b}`
  Both tensors must be the same rank if sizes differ we will 0 pad.

- $\star$(overloaded)
  Tensor product. Examples: `T_{a}*T_{a}`, `T_{a, b, c} * T_{a}`
  Each index to be multiplied over must repeat, these must be the same dimension.
  Scalar product. Examples: `4*8.2`, `a*b`, `T_{0, 1}* X_{10}`
  Both must be scalars
  Scalar-Tensor product. Examples: `4*T_{a}`, `X_{b, c}*T_{0}`
  One must be a scalar, one must be a tensor, order doesn't matter

- ^
  String concatenation. Examples: `"hello," ^ " world!"`
  Both must be strings

- /
  Scalar division. Examples: `3/5`, `a/b`, `T_{0}/T_{1}`
  both must be scalars, the divisor must be non-zero

4.3 Assignment Operator

The single equals (=) is designated as the assignment operator. Assignment is mostly by reference, but sometimes by value. Only primitive types (i.e., float), are assigned by value. All other types (i.e., tensor and string) are assigned by reference. During assignment, the types of the left hand side and right hand side must match. Assignments are expressions, and return the newly assigned value of the left hand side. For a successful assigment, the type of the right hand side must match the type of the left hand side. For more detailed rules of type checking see the Types section.

4.4 Relational Operators

Operator	Description	Example
<	scalar less than	`a < b`
<=	scalar less than or equal to	`a <= b`
>	scalar greater than	`a > b`
>=	scalar greater than or equal to	`a >= b`
==	equal to	`a == b`
~=	not equal to	`a ~= b`

LaTenS does not allow strings or tensors to be compared with any less-than or greater-than operators; only floats can be compared this way because less-than and greater-than requires some total order on the objects being compared. Floats are compared in the expected way, using their numerical value.

Any type of data can be compared with the two equality operators, as long as both operands are of the same type (this means that tensors must be of the same rank). Tensors and strings are compared by reference; floats are compared by value.

An expression involving relational operators is intended to evaluate to a boolean value. Since we do not implement a boolean type, and we instead consider the empty string and scalars equal to exactly `0.0` to be false, and all other values to be true, relational expressions must actually return the scalar `0.0` for false or the scalar `1.0` for true. Use them wisely.

4.5 Order of Evaluation

```
%nonassoc NOELSE
%nonassoc ELSE
%right ASSIGN
%left OR
%left AND
```

```
%left EQ NEQ
%left LT GT LEQ GEQ
%left PLUS MINUS
%left TIMES DIVIDE
%right NOT NEG
```

We follow the example of MATLAB when considering operator precedence. Our operators listed from highest to lowest precedence are as follows:

1. Parentheses ()
2. Get element of tensor (`T_{i, j, ...}`)
3. Unary plus and minus (+, −), logical NOT (~)
4. Tensor multiplication (*), scalar multiplication and division (*, \), tensor-scalar multiplication (*)
5. Addition and subtraction (+, −), string concatenation (^)
6. Relational operators: less than (<), less than or equal to (<=), greater than (>), greater than or equal to (>=), equal to (==), not equal to (~=)
7. Logical AND (&&) and OR (||)
8. Assignment (=)

The assignment operator, the logical NOT operator, the unary minus, and the underscore associate from the right. All other operators are left-associative.

5 Structure

5.1 Statements

```
stmt: expr SEMI                              { Expr $1 }
  | RETURN SEMI                              { Return Noexpr }
  | RETURN expr SEMI                         { Return $2 }
  | LBRACE stmt_list RBRACE                { Block(List.rev $2) }
  | IF LPAREN expr RPAREN stmt %prec NOELSE { If($3, $5, Block([])) }
  | IF LPAREN expr RPAREN stmt ELSE stmt    { If($3, $5, $7) }
  | FOR LPAREN expr_opt SEMI expr SEMI expr_opt RPAREN stmt
                                                          { For($3, $5, $7, $9) }
  | WHILE LPAREN expr RPAREN stmt           { While($3, $5) }
```

A statement is either an expression or some compound statement of expressions. An expression evaluates to a value, meanwhile a compound statement of expressions has no associated value and only defines a side effect. Every statement must be terminated with a semicolon or enclosed in braces. A statement is the smallest unit of computation, some examples of expressions are:

```
[2, 4, 5];
```

The above value expression has no side effect, but evaluates to an immediate value of $[2, 4, 5]$, a single dimensional tensor (a vector).

```
let v_{i} = [2, 4, 5];
```

The above expression defines a vector variable (single dimensional tensor) and assign it the immediate value of $[2, 4, 5]$. The above value expression has both a side effect and an evaluation of $[2, 4, 5]$.

```
s = [2, 4, 5] * [2, 4, 5];
```

The above value expression performs tensor multiplication between two vectors, $[2, 4, 5]$ and $[2, 4, 5]$, and assigns the resultant value to the variable s. This will assume that the first index of each is the one summed over, for vectors this looks like a dot product. The value expression also evaluates to the newly assigned value of s, which in this case is 45.

```
v = foo([2, 4, 5]);
```

The above value expression calls the function foo with a single argument $[2, 4, 5]$. The above assumes that foo is a function taking a single argument, a one dimensional lower tensor, and returns a value. The syntax and semantics of functions will be covered in a later section.

Compound statements that do not evaluate to a value are typically found in code flow constructs, and allow the flexibility of evaluating to no value in situations where there is no suitable candidate evaluation value. Some common void expressions are:

```
if (evalexpr)
{
    stmt1
    stmt2
    ...
    stmtn
};
```

The entire if block above is a statement that evaluates to no value, this means the following is illegal:

```
v = if (evalexpr)
    {
    ...
    };
```

The above is illegal because *void* is not a value, but rather a guarantee that *no* value exists. Another common void expression is found in loops:

```
for(beginstmt; evalexpr; iterstmt)
{
    stmt1
    stmt2
    ...
    stmtn
};
```

Paradoxically, function evaluations can both be an expressions and statement. For example the following may be illegal:

```
v = bar([2, 4, 5]);
```

This depends on the *type* of a function, which may be void.

Compound statements are limited to loops, if-then-else, expression sequences, and *void* functions.

5.2 Expression

An expression is one of assignment, unary or binary operators applied to expressions, or a *non-void* function evaluation.

5.3 Statement

A Statement is one of statement sequences, if-then-else, for loop, or while loop.

5.4 Block Statement

A block statement is a void expression, which comprises of zero or more statements. The syntax for a block statement is as follows:

```
{
    stmt1;
    stmt2;
    ...
    stmtn;
}
```

5.5 True and False

Boolean types do not exist in LaTenS. There are analogues, however. In particular, only an empty string and a zero dimensional tensor (a scalar) with the value of $+0$ or -0 evaluates to False. All other values evaluate to True.

5.6 If-then-else

An if-then-else comes in two following forms:

```
if(evalexpr)
stmt1
```

```
if(evalexpr)
stmt1
else
stmt2
```

The semantics of the above is that if evalexpr is evaluated to *True*, then the if branch is executed. If an else follows the if, then it is executed only if evalexpr evaluates to *False*. The execution of *stmt1* and *stmt2* is mutually exclusive.

5.7 Loops

The syntax of for loop in LaTenS is as follows:

```
for(beginstmt; evalexpr; iterstmt)
stmt
```

The semantics of the above is to execute *beginstmt* in the prologue, and continuing to execute the loop body while *evalexpr* evaluates to *True*. To be more specific the loop body consists of *stmt* followed by *iterstmt*. Prior to any execution of the loop body, *evalexpr* must be evaluated and checked to be True.
The syntax for while loops is as follows:

```
while(evalexpr)
stmt
```

The semantics is to execute *stmt* as long as *evalexpr* evaluates to True.

5.8 Function definition

```
fdecl: typ ID LPAREN formals_opt RPAREN LBRACE stmt_list RBRACE
       { { typ = $1; fname = $2; formals = $4;
           body = List.rev $7 } }
```

The returntype may either be *void*, or some other concrete type.

5.9 Scoping

Functions and expression sequences intimately interact with *scope*. Abstractly, a *scope* is a set of bindings between identifiers and their underlying meaning. In particular each expression sequence defines a container of mappings of identifiers to their meaning (a scope).
During the evaluation of an expression sequence each *assignment* statement adds a mapping from the identifier to its meaning in the container corresponding to the current scope. Because expression sequences can nest, and there are many such containers each having possibly conflicting mappings. Therefore there are strict rules for evaluating the meaning of an identifier based on commonly accepted scoping rules.
Define the innermost scope as the mapping belonging to the currently executing expression sequence. Define innermost-0 as innermost. Define innermost$-i$ as the mapping belonging to an expression sequence which contains an expression sequence statement using innermost$-i+1$ as its scope. Define outermost as any scope not contained in any other scope. Given the above definition the following rules are followed for mapping:

```
map(id)
{
    let i = 0
    while(innermost-i != outermost)
    {
        if(innermost-i.contains(id))
        {
            return innermost-i[id];
        }
        i++;
    }
}
```

Function definitions also subtly interact with scope in a slightly different manner. In particular, a function definition adds the function to the *outermost* scope. Furthermore, this addition is performed prior to the execution of the statement associated with the function definition. This allows a function to be in its expression sequence's scope. This is necessary for recursion. Note the above rules do not allow forward declarations. In particular mutually recursive functions are not allowed in LaTenS.

5.10 Program

```
program: decls EOF { $1 }
```

A program is a sequence of function definitions ending with a specially named function called *main*:

```
returntype1 functionname(argtype11 arg11, argtype21 arg21, ..., argtypen1 argn1)
```

```
stmtlist1

returntype2 functionname(argtype12 arg12, argtype22 arg22, ..., argtypen2 argn2)
stmtlist2

...

returntypem functionname(argtype1m arg1m, argtype2m arg2m, ..., argtypenm argnm)
stmtlist3

void main()
stmtlistmain
```

The main function follows all previously defined rules of functions, and is the entry point of the LaTenS program. Program execution begins at the first statement in the exprseqmain.

References

[1] http://www.openicon.com/mu/math/tensors_01.html

[2] https://www.gnu.org/software/gnu-c-manual/gnu-c-manual.html#Real-Number-Constants

[3] https://www.gnu.org/software/gnu-c-manual/gnu-c-manual.html#String-Constants