# LaTenS: A Tensor Manipulation Language


| Daniel Schwartz | Eliana Ward-Lev | Mohit Rajpal | Elsbeth Turcan |
|---|---|---|---|
| *Manager* | *Language Guru* | *System Architect* | *Tester* |
| ds3263 | erw2138 | mr3522 | ect2150 |


September 28th, 2016

## 1 Introduction

### 1.1 Background

Tensors are a generalization of vectors or matrices to higher dimensions. A zeroth order tensor is a scalar, a first order tensor is a vector, a second order tensor is a matrix and so on.

$T^c_{ab}$ is the notation for a tensor named T with 2 down indices and one up index. The latex notation, T_ab^c, notation is used to name our variables below.

### 1.2 Motivation

The name LaTenS is a combination of LaTeX, a typesetting language, and the word "tensor", the mathematical object that will be at the core of our language. Our motivation in writing LaTenS is to generalize tensor calculations and make them easier and more efficient for scientists studying fields such as general relativity, which make extensive use of tensors. As such, it can be expected that tensors will be primitives and their notation will reflect the current standard in LaTeX. Additionally, all basic tensor operations will be easy to use. It is further expected that users will write libraries to implement the standard functions utilized in their fields.

Note: While there are currently languages built for matrix mathematics (ex: Python w/ Numpy), and even languages that enable tensor mathematics (ex: MATLAB), to the best of our knowledge, there are no languages that treat them as primitives.

## 2 Syntax and Features

### 2.1 Types

LaTenS is an untyped language. We support three types of declarations, however: an $n$th-order tensor in which the elements are a generic "numeric" type (that is, floating point), a scalar (which is really just a 0th-order tensor), or a string. Any variable can be evaluated as a boolean.

| Name | Description | Example Declaration |
|---|---|---|
| Tensor | $n$th-order tensor; use _ to define lower indices and ^ to define upper indices | `T_mn = [[1, 0], [0, 1]];` |
| Scalar | Special shorthand for a scalar | `a = 5.5;` |
| String | A string | `str = "String Text";` |

Note that a scalar can be declared with this special shorthand, but it will be stored and manipulated as a 0-dimensional tensor.
Any tensor or string can be evaluated as a boolean value. A tensor with all values set to `0`, or the empty string, are considered false for the purposes of boolean logic; all other values are considered to be true.

## 2.2 Operators

LaTenS will support several useful built-in operations on tensors. (Strings are primarily meant for printing information, and therefore we support only string concatenation as an operation on strings.) Whitespace is naturally ignored when evaluating expressions, so spaces are allowed between terms. Relational operators apply mainly to scalars (that is, we will need to verify that the operands are 0th-order tensors and not strings or other tensors), while the quality operators can be applied to any operands.

### 2.2.1 Binary Operators

| Operator | Description | Example |
|---|---|---|
| `+` | tensor addition | `T_a+T_b, T_a + T_b` |
| `-` | tensor subtraction | `T_a-T_b, T_a - T_b` |
| `*` | tensor product | `T_a*T^a, T_a * T^a` |
| `strcat(String1, String2)` | string concatenation | `strcat("Hello ", "world")` |
| `<` | scalar less than | `a<b, a < b` |
| `<=` | scalar less than or equal to | `a<=b, a <= b` |
| `>` | scalar greater than | `a>b, a > b` |
| `>=` | scalar greater than or equal to | `a>=b, a >= b` |
| `==` | equal to | `a==b, a == b` |
| `~=` | not equal to | `a~=b, a ~= b` |

Note that tensor division is properly defined as multiplication by the inverse, and so is not supported as a basic operator.

### 2.2.2 Unary Operators and Standard Functions

| Operator | Description | Example |
|---|---|---|
| `Inv(Tensor)` | inverse | `Inv(T^a)` |
| `raise(Tensor, Index)`, `lower(Tensor, Index)` | promote or demote index $n$ | `raise(T_a,a)`, `lower(T_a^mn, m)` |
| `dim(Tensor)` | get dimensions of tensor | `dim(T_a)` |
| `dim(Tensor, Index)` | get size of a particular dimension | `dim(T_mn, m).` |
| `_, ^` | get element by indices | `T_00` |

## 2.3 Control Flow

LaTenS supports basic control flow such *if–then–else* and *for* loops.

### 2.3.1 *if–then–else*

*If–then–else* is an expression that returns *void*, but executes only one branch. An *if–then–else* expression must be of the following format.

```
if (expr) {
    expr1_True
    expr2_True
    ...
}
else {
    expr1_False
    expr2_False
    ...
}
```

As it is obvious by the syntax, *expr* is evaluated. If *expr* is a tensor of one or more dimensions, it is assumed to be *True*. If *expr* is a 0 dimensional tensor [a scalar] and is equal to $-0$ or $+0$ in IEEE 754 Floating Point representation, *expr* evaluates to False. Likewise, if *expr* is a string and is the empty string, *expr* evaluates to False. Otherwise *expr* evalutes to True. If *expr* is type void then it evaluates to False.

### 2.3.2 Loops

For succinctness only *for* loops are supported in C99 style. The return type for for loops is void.

```
for(beginexpr; evalexpr; iterexpr)
{
    expr1_Loop
    expr2_Loop
    ...
}
```

The evaluation of a for loop begins with executing *beginexpr*. The loop body execution begins with checking *evalexpr* is True in the same style as *if–then–else* expression. The expressions inside the loop body are executed in sequence. Afterwards the *iterexpr* is evaluated and executed.

### 2.3.3 Functions

LaTenS supports simple functions with return statements.

```
functionName(varname1, varname2 ... varnamen)
{
    expr1_Fun
    expr2_Fun
    ...
    return exprm_Fun
}
```

Latens does no compile time checks on argument types or return types. A function may take or return any expression. It is the caller's and callee's responsibility to ensure the semantics fo functions.

## 2.4 Comments

Comments are parsed in a regular expression format. LaTenS supports single line comments

```
% Single Line Comment
```

and block comments over multiple lines

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

Nested comments are not supported.

## 2.5 Features and Design Decisions

LaTenS allocates memory on demand. Users do not need to worry about memory management. For the sake of simplicity, LaTenS does minimum allocation on stack. The stack is used for LaTenS internals, and all objects exist on the heap. This saves complexity as the LaTenS compiler does not need to determine whether an object should be *statically* or *dynamically* allocated.
LaTenS lacks exoctic features such as recursion, type checking, and automatic garbage collection. Memory is automatically allocated, but LaTenS follows the Ostrich philosophy. The ostrich sticks its head in the sand at the first sign of danger. Just so, LaTenS leaks memory like the Titanic takes on water.

Pointers and references are not allowed in LaTenS. The authors of the LaTenS language believe in strict gun control and do not wish the users of LaTenS to shoot themselves in the foot.
LaTenS does minimal safety checks during compile time, all errors and issues are reported to the programmer during runtime. This saves effort by not attempting to prove invariants regarding the code.

# 3 Sample Program

In General Relativity, problems often involve tensors. A simple problem is determining if a four-vector is space-like or time-like. This would involve multiplying two one-dimensional tensors and one two-dimensional tensor. In LaTenS this would take the form

```
g_mn = [[1, 0,  0,  0],
        [0, -1, 0,  0],
        [0,  0, -1, 0],
        [0,  0,  0, -1]]
%{g now is the Minkowski metric which defines space-time in flat space. g is a global variable
    which can be used but not changed.}%

squared(u^a, name) {
    print(name);
    return u*u*g;
}

isSpaceLike(ip, name) {
    print(name);
    if(ip > 0){
        print(" is space-like");}
    else if(ip = 0){
        print(" is light-like");}
    else { %ip < 0
        print(" is time-like");}
    }
}

main() {
    u^a = [5, 1, 1, 0];
    isSpaceLike(squared(u, "u"));
    v^a = [1, 1, 1, 1];
    isSpaceLike(squared(v, "v"));
    isSpaceLike(squared(u+v, "u+v");
}
```

The program would print

```
u is space-like
v is time-like
u+v is space-like
```