Guide to TenFlang

This document will describe the principles and syntax of TenFlang (TENsorFlow Language), and how the Preprocessor expands TenFlang program so they can be compiled and interpreted. We will first describe the large ideas of the language, then delve deeper into each type of instruction. We will examine the rules and syntax of each instruction type, and how the Preprocessor is responsible for expanding instructions of each type.

Overview of the Language

At a high level, a TenFlang program consists of variable declarations and variable definitions. Every variable in the program must be declared before it is defined. A variable declaration establishes the name and type of a variable. A variable definition assigns a value to a variable. This value is an expression involving other previously defined variables and/or constants. The TenFlang language supports seven primitive operations: addition, subtraction, multiplication, exponentiation, natural logarithms, base-e exponentation, and the logistic function. With these seven primitive operations, just about any basic mathematical operation can be expressed (in fact, the logistic function can be built up from the other primitives).

TenFlang programs are straight-line programs; that is, there are no control-flow statements. These programs are interpreted in one pass, line by line, from top to bottom, so there are no forward references, loops or recursion.

There are two more advanced features of TenFlang, vectors and user-defined macros. Vector notation is extremely useful when a program involves data best represented by an array or list. For example, a program's input may be a photo represented as a 28-by-28 array of pixels, or a flattened 784-element pixel vector. Vector notation allows the writer of the program to declare and perform mathematical operations on this pixel vector, and conveniently abstracts away the underlying primitive component-wise operations. TenFlang supports vector declarations, and vector operations, like dot products or component-wise sums. It is the job of the Preprocessor to then expand a vector declaration or operation into component-wise instructions that the Compiler or Interpreter knows how to handle.

As previously mentioned, variables are defined as mathematical expressions involving operations on other variables and/or constants. Recall that almost all computations can be built up from our seven primitive operations. However, the writer of a program may find that there is a complex mathematical function that he wishes to use repeatedly. At the top of his program, the writer can define a macro, which states how the operands are manipulated through a chain of primitive operations to arrive at the result. In subsequent lines, the writer can then simply define a variable as the result of applying his macro to the other variables, rather than writing out the complex function repeatedly. It is the job of the Preprocessor, then, to expand each reference to the macro into the primitive operations of which it consists.

For example, suppose a program uses a sum-of-squares operation 100 times. To sum the squares of two variables involves three primitive operations: squaring both variables individually, then performing the addition. Defining the macro once:

```
#macro c = sum_of_squares a b; declare intvar p; define p = pow a 2;
declare intvar q; define q = pow b 2; define c = sum p q;
```

would save about 500 lines of user-written code, as compared to writing out the expanded version 100 times. The macro concept is motivated by the same principles of modularity that motivate writing functions in other major programming languages.

We will now take a more detailed look at all the types of instructions.

The Five Types of Instructions

TenFlang supports five types of instructions, *declare*, *define*, *declare_vector*, *define_vector*, and *#macro*.

The DECLARE Instruction

Every variable used in a TenFlang program must be declared, and no variable can be declared twice. A declaration must be of the format:

```
declare <variable type> <variable name>
Ex. "declare input x"
```

There are six types of variables:

- *input*
  - An input to the computation.
- *exp_output*
  - An expected value for a given input. Expected outputs are typically involved in calculating the loss function.
- *weight*
  - A free variable, the value of which is not initially known to the user. The ultimate goal of using this system is to compute the weights.
- *intvar (intermediate variable)*
  - An intermediate variable in the computation.
- *output*
  - The observed output of a program. Outputs are a function of inputs and weights.
- *loss*
  - An expression that measures the deviation of the output from the expected output.

A valid variable name in TenFlang cannot be a **keyword**. A keyword is the name of an instruction (*declare, define, declare_vector, define_vector, #macro*), the name of an operation (*add, sub, mul, logistic, exp, pow, ln, dot, reduce_vector*), or the name of a variable type (*input, output, weight, intvar, exp_output, loss*). Additionally, a valid variable name must not contain numerals. This is to ensure there are no naming ambiguities with the intermediate variables generated when the Preprocessor expands instructions involving vectors or macros. These Preprocessor-generated variables follow naming conventions involving names with numerals in them.

The DECLARE_VECTOR Instruction

Every vector used in a TenFlang program must be declared, and no vector can be declared twice. A vector declaration must be of the format:

```
declare_vector <vector type> <vector name> <vector dimension>
Ex. "declare_vector weight w 3"
```

The vector types are the same as the variable types mentioned in the previous section. Every individual element in the vector is to be a variable of this type. As with the *declare* instruction, vector names cannot be keywords, and cannot have numerals in them. The vector's dimension must be a positive integer.

The Preprocessor expands a *declare_vector* instruction into as many simple *declare* instructions as there are elements in the vector. For example, `"declare_vector weight w 3"` is expanded into:

```
declare weight w.0
declare weight w.1
declare weight w.2
```

The DEFINE_VECTOR Instruction

The *define_vector* instruction defines a vector as a manipulation of one or two other vectors. This instruction defines all the elements of the result vector at once. A vector definition must be of the format:

```
define_vector <vector_name>  =  <operation (primitive/macro)>
<operand 1 (vector)> <optional operand 2 (another vector, a variable,
or a constant)>
```

`Ex. "define_vector z = mul x y"`
(Assume that *z* was previously declared, and that *x* and *y* were both previously declared and defined. For the rest of this section, assume *p* is a (scalar) variable that equals 3, and *x* and *y* are vectors, such that `x = {1, 2, 3}` and `y = {4, 5, 6}`.)

The operands to a *define_vector* instruction are either:

- Two vectors of the same dimension
  - The *i*-th element of the result vector is the result of the binary operation being applied to the *i*-th element of both operand vectors.
  - `"define_vector z = add x y" → z = {5, 6, 7}`
- One vector and one (scalar) variable
  - The *i*-th element of the result vector is the result of the binary operation being applied to the *i*-th element of the operand vector and the scalar variable.
  - `"define_vector z = mul x p" → z = {3, 6, 9}`
- One vector and one constant
  - The *i*-th element of the result vector is the result of the binary operation being applied to the *i*-th element of the operand vector and the constant.
  - `"define_vector z = pow x 2" → z = {1, 4, 9}`
- One vector
  - The *i*-th element of the result vector is the result of the unary operation being applied to the *i*-th element of the operand vector.
  - `"define_vector z = my_unary_macro x" → z = {my_unary_macro(1), my_unary_macro(2), my_unary_macro(3)}`

To be valid, a *define_vector* instruction must meet these criteria:

- The vector being defined must have been previously declared, and cannot have been previously defined.
- The vector being defined cannot be of the type *input, weight*, or *exp_output*. The values of variables of these three types are passed as inputs to the TenFlang interpreter; thus, it does not make sense to define an *input*, *weight*, or *exp_output* variable or vector.
- The first operand vector must have been previously declared and defined. (If this operand vector is an *input*, *weight*, or *exp_output* vector, it must have been previously declared.)

- If the primitive/macro operation is a binary operation, and the second operand is a vector or a variable, it too must have been previously declared and defined. (As before, if this operand is an *input*, *weight*, or *exp_output* vector/variable, it must have been previously declared.)
- The result vector must be of the same dimension as the first (and possibly second) operand vectors. If both operands are vectors, they must be of the same dimension.

The Preprocessor expands a *define_vector* instruction into as many *define* instructions as there are elements in the vector. For example, "`define_vector z = add x y`" is expanded into:

```
define z.0 = add x.0 y.0
define z.1 = add x.1 y.1
define z.2 = add x.2 y.2
```

The MACRO Instruction

TenFlang program writers are allowed to define their own functions called *macros*. Macros take in one or two values, and output one value. The definition of a macro must be of the format:

```
#macro <dummy result variable name> = <macro name> <first operand
dummy name> <(optional) second operand dummy name>;
... simple declare or define instructions separated by semi-colons
...
; define <dummy result variable name> = <some expression>;
```

`Example Binary Macro` (Computes the sum of the squares of the two inputs):

```
#macro c = sum_of_squares a b; declare intvar p; define p = pow a 2;
declare intvar q; define q = pow b 2; define c = add p q;
```

`Example Unary Macro` (Computes $\frac{1}{x+3}$ for an input x):

```
#macro foo = my_unary_macro bar; declare intvar baz; define baz = add
bar 3; define foo = pow baz -1;
```

The point of a macro is that it encapsulates several lines of code and gives this bundle one name. The intermediate *declare* and *define* instructions typically declare and define *intvars* as the result of intermediate operations on the macro's inputs. No other types of variables can be declared or defined in the body of a macro. The last line of the macro defines the output of the macro, and is typically a function of the inputs and/or intermediate variables. As before, the names of the macro's result and operand parameters, as well as any intermediate variables, must be valid variable names (no keywords or names with numerals). All macro definitions must appear at the top of the program.

The Preprocessor parses a *macro* instruction and stores the necessary data so that when the macro is referenced later in the program, it can accurately substitute in the body of macro (see following section on *define* instructions for how the Preprocessor expands a reference to a macro in a *define* instruction).

The DEFINE Instruction

The *define* instruction assigns a value to a scalar variable. The value is one of the following:

- A constant
  - `define <variable name> = <constant>`
- The value of another variable
  - `define <variable name> = <another variable name>`
- The result of a binary operation (primitive or macro) on two quantities (variables and/or constants)
  - `define <variable name> = <binary operation> <variable/constant operand 1> <variable/constant operand 2>`
- The result of a unary operation (primitive or macro) on one quantity (a variable or a constant)
  - `define <variable name> = <unary operation> <variable/constant operand>`
- The result of the dot product of two vectors
  - `define <variable name> = dot <first vector operand> <second vector operand>`
- The result of "reducing" a vector (applying an operation to all the elements of a vector in a cascading fashion, from left to right)
  - `define <variable name> = reduce_vector <vector operand> <binary operation (primitive or macro)>`

Here is a list of various examples of *define* instructions:

```
define z = 3
define z = x
define z = add x y
define z = mul x 3
define z = my_binary_macro x -4
define z = exp x
define z = ln 1
define z = my_unary_macro 2
define z = dot u v
define z = reduce_vector v add
```

- defines *z* as the sum of all the elements in the vector *v*

```
define z = reduce_vector u my_binary_macro
```

- defines *z* as my_binary_macro(u_n , my_binary_macro(u_{n-1} , my_binary_macro(u_{n-2} , ... , my_binary_macro(u_1 , u_0))))

To be valid, a *define* instruction must meet these criteria:

- The variable being defined must have been previously declared, and cannot have been previously defined.
- The variable being defined cannot be of the type *input, weight*, or *exp_output*. The values of variables of these three types are passed as inputs to the TenFlang interpreter; thus, it does not make sense to define an *input*, *weight*, or *exp_output* variable.
- Any operands that are variables or vectors must have been previously declared and defined. (If this operand is an *input*, *weight*, or *exp_output*, it must have been previously declared.)
- If the instruction involves a dot product, both the operand vectors must be of the same dimension.

Preprocessor Expansions of DEFINE instructions

The Preprocessor must expand *define* instructions if they involve macros, dot products, or vector reductions.

If a variable is defined as the result of a macro, the body of the macro must be substituted into the expanded program. This is best described with an example. Suppose the *sum-of-squares* binary macro has been defined as such:

```
#macro c = sum_of_squares a b; declare intvar p; define p = pow a 2;
declare intvar q; define q = pow b 2; define c = add p q;
```

Now consider the following *define* instruction. Assume that *z* has been previously declared, and that *x* and *y* have been previously been declared and defined. Assume also that this is the first reference to the *sum-of-squares* macro in this program.

```
define z = my_macro x y
```

This gets expanded as follows:

```
declare intvar z_p:0
define z_p:0 = pow x 2
declare z_q:0
define z_q:0 = pow y 2
define z = add z_p:0 z_q:0
```

Note the following important points:

- The intvars in the body of the macro are given a canonical name; the name *z_p:0* indicates that this variable is the intvar *p* in the expansion of the macro used to define the variable *z*, and that this is the 0-th (first) reference to the *sum-of-squares* macro. It is impossible for the Preprocessor ever to generate a variable with the same name later in the program.

- The arguments to the macro, *x* and *y* are substituted in place of *a* and *b*, the dummy parameter names in the initial macro definition. *z* is substituted in place of the dummy result parameter *c.*

If a variable is defined as the dot product of two vectors, the Preprocessor expands this instruction to a series of component-wise additions and multiplications. This is best illustrated with an example. Consider the following *define* instruction. Assume that *z* has been previously declared, and that *u* and *v* are 3-element vectors that have been previously been declared and defined.

```
define z = dot u v
```

This gets expanded as follows:

```
declare intvar z.0
define z.0 = mul u.0 v.0
declare intvar z.1
define z.1 = mul u.1 v.1
declare intvar z.2
define z.2 = mul u.2 v.2
declare intvar z.3
define z.3 = add z.0 z.1
define z.4 = add z.3 z.2
define z = z.4
```

If a variable gets defined as the result of a vector reduction, the Preprocessor expands this instructions into a series of intermediate operations. The given operation is applied to the 0th and 1st elements of the vector, then the operation combines this intermediate value with the 2nd element of the vector, then with the 3rd, and so on. The expansion is best illustrated with an example. Consider the following *define* instruction. Assume that *z* has been previously declared, and that *u* is a 3-element vector that has been previously been declared and defined.

```
define z = reduce_vector u add
```

This gets expanded as follows:

```
Declare intvar z.0
Define z.0 = add u.0 u.1
declare intvar z.1
define z.1 = add z.0 u.2
Define z = z.1
```