# Synapse Language Project Report

Jonathan Williford
jw2389 @ columbia.edu

http://synapse-lang.googlecode.com

# Contents

# Chapter 1

# Introduction

Before mathematical models were used in neuroscience, models have mainly been limited to imprecise word models. Such word models that have sounded reasonable in the past have turned out to be inconsistent and unworkable when trying to convert to a mathematical model [Abbott]. Simulation enables precise models to be tested on large interconnected networks. The proposed language Synapse is a language specifically for modeling and simulating neural networks.

While every neuron in the brain executes in parallel, most languages are written for architectures that execute sequential. Even as parallel computing becomes more important, parallel support is usually added as an afterthought. For example, CUDA relies on extending C and C++ so that it can take advantage of nVidia's graphic cards and OpenMP adds C preprocessor commands to enable, among other things, parallel for-loops. Synapse is a language that being created for parallel execution from the ground up.

The source code and documentation (including the LaTeX source for PDFs) may be downloaded from `http://synapse-lang.googlecode.com`.

# Chapter 2

# Language Tutorial

## 2.1 Introduction

The Synapse language is currently implemented as an interpreter, however future versions may be able to compile C/C++ with OpenCL or CUDA and MATLAB. Synapse was designed for programs that simulate biological neural networks and run in a massively concurrent manner.

## 2.2 Installing

The code for the interpreter can be downloaded from `http://synapse-lang.googlecode.com`. You will also need to download and install Ocaml `http://caml.inria.fr/ocaml/`. Once you have done this, you can go into the "src" directory and type "make". To test the install, you can type "make test". The output will be very verbose, however a summary of the results will be shown at the end.

## 2.3 Writing your first Synapse programs

The simplest program is following:

```
input $1[5]; /* Define the input parameter as a vector with 5 elements */

$2 << $1; /* Copy input to output */
```

Before running the program you first need to create the input file. One of the formats that Synapse allows is space-delimited files. Each line in the file corresponds to a single time step. Here is an example that works with our previous example:

```
1 2 3 4 5
0 1 0 1.2 0
7 .8 3 2 2
2 8 .9 3 0
```

You can run the example with:

```
.\synap input.txt output.txt < program.syn
```

Synapse allows you to flip the input quickly. For example, the following could mirror an image with:

```
input $1[480,640,3]; /* Height, width, and number of channels for an image */
$2[ y,size($1,2)-x+1,c] << $1[y,x,c] for x=[1:640] y=[1:480] c=[1:3];
```

## 2.4 Activation Functions

Functions are helpful in order to do more complex tasks. There are two types of functions allowed in Synapse. Activation functions take a single float and returns a single float. Optional parameters can also be passed. The following example performs gamma correction on 128x128 images.

Example ex-gammacorrection.syn

```
input $1[128,128,3];

gammaCorrection( x; gamma=2.2 ) = x ^ (1/gamma);

$2[y,x,c] << gammaCorrection( $1[y,x,c]; gamma=1.5) for x=[1:128] y=[1:128] c=[1:3];
```

(a) Original image (b) Resulting image

Figure 2.1: An example input and output image resulting from Example ex-gammacorrection.

Please note that the ability to read in sequences of images (in PPM format) is very experimental and known bugs exist, for example, the file format for the output images is currently ignored. This example can be found in the tests directory in the subversion repository. It can be run with "./run_examples.sh ../tests/ex-gammacorrection.syn" in the subversion src directory.

## 2.5 Kernel Functions

The other type of functions that are allowed are kernel functions. Kernel functions can be applied to (or convolved with) matrices (or expressions that results in matrices). The are defined with the keyword **kernel**. There is a non-optional parameter for each dimension of the expression for which it is convolved.

Example test-kernel1d.syn

```

input $1 [ 5 ];

kernel foo(i) = 1/(i*i+1);

$2 << $1 ** foo();
```

## 2.6 Modules and Neurons

In order to be able to model biological neural networks, the network connections need to be represented. This can be done with the use of modules and neurons. Every neuron must be in a module, which could be equated to a hypercolumn or micronetwork. A module specifies input neurons, output neurons, and inter-neurons. The input and output neurons are the only neurons that can have external connections. The below example only uses input and output neurons.

Example test-module1.syn

```

input $1 [ 5 ];

module half in[5] >> out[5]
{
   out[i] << in[i] / 2 for i = [1:5];
}

half.in[j] << $1[j] for j = [1:5];

$2[k] << half.out[k] for k = [1:5];
```

Each synapse connection is evaluated concurrently, which means that it can take several or many time steps for the values to propagate through the network, but it also means that it can easily utilize multi-core architectures.

See Appendix B for more a lot more examples and the next chapter for the rules of the Synapse language.

# Chapter 3

# Language Manual

## 3.1 Document Conventions

Literals are denoted with **monospace**. Syntatic categories are denoted with *italics* and are all lowercase. Identifiers, integers, and floats are represented by *Id*, *Int*, and *Float* respectively. Optional items are indicated with "opt" in subscripts following the item, ex. $\textit{optional-item}_{opt}$ . Sometimes the syntatic categories are enumerated in the suffix, ex. *item-1*, for ease of reference. Section numbers to the right of the productions indicate the location of syntatic categories not defined in the same subsection.

## 3.2 Lexical Conventions

### 3.2.1 Comments

Comments begin with the characters /* and continue until */.

### 3.2.2 Identifiers

Identifiers consist of letters, digits, and underscores. The first character must be a letter. Identifiers are case sensitive.

### 3.2.3 Keywords

The following identifiers are reserved keywords and may not be used for any other purpose.

| | |
|---|---|
| **module** | |
| **size** | Checked with §B.12 test-flip5a. |
| **for** | |
| **t** | |
| **end** | |
| **kernel** | |
| **pi** | Checked with §B.6 test-constant-pi. |
| **e** | Checked with §B.5 test-constant-e. |
| **sin** | Checked with §B.32 test-sin. |
| **cos** | Checked with §B.10 test-cos. |
| **exp** | Checked with §B.11 test-exp. |
| **pragma** | |
| **input** | |

**t** and **end** are not used but are reserved for later[1].

[1] Checked with §B.44 fail-reserved-word-t and §B.45 fail-reserved-word-end.

### 3.2.4 Constants

There are two types of constants, int constants and float constants.

#### Integer Constants

An int consists of one or more digits.

#### Float Constants

A float consists of a decimal point and at least one digit. The precision of the float is compiler dependent and may even be implemented as an integer using scaling. An int can be implicitly casted as a float, but not vice versa.

#### Built-in Constants

*e* and *pi* are built-in constants which are approximately 2.71828 and 3.14159 respectively[1]. The accuracy depends on the precision of float used by the compiler.

### 3.2.5 Program Parameters

The input and output sources of the program are specified by $1, $2, etc. For command-line applications $1 corresponds to the first parameter, $2 the second, etc. How they are used will determine whether they are input or output. They may not be both [2]. Every input must be declared with its dimension.

*input-decl*:
  **input** *Param dimensions* **;** §3.3.3

## 3.3 Program

A program consists of module definitions, kernel function definitions, activation function definitions, and synaptic connections.

*program*:
  */* nothing */*
  *input-decl program* §3.2.5
  *module-def program* §3.3.1
  *activation-def program* §3.3.2
  *kernel-def program* §3.3.2
  *synap-connection program* §3.3.3

### 3.3.1 Module Definition

Neurons can only be defined in modules. There are three exclusive types of neurons in a module: input neurons, output neurons, and inner neurons. Input neurons receive external signals, output neurons send external signals, and inner neurons are encapsulated in the module.

*module-def*:
  **module** *Id neurons-1* >> *neurons-2* **{** *module-body* **}** *Id* is the name of the module. *neurons-1* and *neurons-2* are the list of input and output neurons respectively.

[1] Checked with §B.5 test-constant-e and §B.6 test-constant-pi.
[2] Checked with §B.36 fail-in-out-param.

*neurons*:
    *Id dimensions*$_{opt}$
    *Id dimensions*$_{opt}$ , *neurons*

The inner and output neurons are defined with an activation expression inside of the module using *neuron-def*. The activation expression of an input neuron is defined by synaptic connections outside of the module using *synap-connection*.

*module-body*:
    */* nothing */*
    *neuron-def module-body* — Within a module
    *synap-connection module-body* — Between modules, §3.3.3

*neuron-def*:
    *Id dimensions*$_{opt}$ << *expression* **;**
    *Id dimensions*$_{opt}$ << *expression* **for** *for-list* **;** — §3.6.1

The variable iterators can only be used in *expression*. *dimensions* is used to specify the size of the array of neurons and must be equal in size of the expression that is being iterated over.

Modules may be used directly:

```
modulename1.input << modulename2.output;
```

or may be instantiated:

```
modulename1 mods[2];
mods[1].input_neuron << mods[2].output_neuron;
```

### 3.3.2 Function Definitions

There are two types of functions allowed in Synapse: activation functions and kernel functions. Activation functions take and returns a scalar while kernel functions generate matrices that fit the context referenced.

#### Activation Function Definitions

Activation functions take a single scalar and returns a single scalar[1].

*activation-def*:
    *Id-1* ( *Id-2 fparams*$_{opt}$ ) = *expression* ;

*Id-1* is the name of the function and *Id-2* is the name of the local input scalar. *fparams* are optional[2] and may be used to define parameters of type float with their default values[3].

*fparams*:
    */* nothing */*
    **;** *fparam-list*

*fparam-list*:
    *Id* = *Float*
    *Id* = *Float* , *fparam-list*

---

[1] Checked with §B.34 fail-afun-mat.
[2] Checked with §B.1 test-afun1.
[3] Checked with §B.2 test-afun2 and §B.3 test-afun3.

#### Kernel Function Definition

Kernel definitions[1] may only be used directly to the right of a convolution operation (§3.4.2).
*kernel-def*:
  **kernel** *Id* **(** *id-list fparams*$_{opt}$ **)** **=** *expression* **;**   §3.3.2,3.4

*id-list* contains the comma-delimited names for the indices that may be referenced in *expression*. The first index refers to the first dimension (the row if 2D), the second index refers to the second dimension, etc. If $w$ is the number of cells in a dimension, then the indices are enumerated from $-\frac{w-1}{2}$ to $\frac{w-1}{2}$ while incrementing by 1. Therefore, if the dimension is even, then the index values will not be an integer.

The Gabor filter can be implemented as:

```
kernel gabor(x,y,lambda=1,theta=0,psi=0,sigma=1,gamma=0) =
   exp(-((x*cos(theta) + y*sin(theta))^2+gamma ^ 2 * (-x*sin(theta)
         + y * cos(theta))^2)/(2 * sigma^2))
 * cos( 2*pi*(x*cos(theta) + y*sin(theta))/lambda+psi);
```

### 3.3.3 Inter-Module Synaptic Connections

The synaptic connections are used to connect the input and output neurons between modules.
*synap-connection*:
  *neuron-scoped dimensions*$_{opt}$ **<<** *expression* **;**   §3.3.3
  *neuron-scoped dimensions*$_{opt}$ **<<** *expression* **for** *for-list* **;**   §3.6.1 **Param** *dimensions*$_{opt}$ **<<** *expression* **;**
  **Param** *dimensions*$_{opt}$ **<<** *expression* **for** *for-list*

*dimensions*:
  **[** *const-int-list* **]**

*const-int-list*:
  *const-int-expr*   §3.4.1
  *const-int-expr* **,** *const-int-list*

See §3.7 for the definition of *neuron-scoped* and for information on scoping. The last two definitions, with **Param**, can only be used if the connection is made in the global scope.

## 3.4 Expressions

The subsections below appear from highest to lowest precedence. Operators within a subsection have equal precedence.

[1] Checked with §B.15 test-kernel1d.

*expression*:
*primary-expression*
( *expr* )
*expr* + *expr*
*expr* - *expr*
*expr* * *expr*
*expr* / *expr*
*expr* ˆ *expr*
- *expr*
*expr* ** *kernel-call* §3.5.2
**pi**
**e**
**exp** ( *expr* )
**sin** ( *expr* )
**cos** ( *expr* )

### 3.4.1 Primary expressions

Primary expressions include the below syntatic category plus kernel function calls. Kernel function calls can only appear to the right of a convolution operator (3.4.2).

*primary-expression*:
*Float*
*Int*
*indexable-expression* $indices_{opt}$
*activation-call* §3.5.2

*indexable-expression*:
*Param*
*Id*
*scoped-neuron*

*indices*:
[ *index-list* ]

*index-list*:
*index-num*
*index-num* , *index-list*

One-based indexing is used. *index-num-1* is the first number in the range when expanded and *index-num-2* is the last. If specified, the middle number, *const-int-expr*, specifies the increment, otherwise each consecutive number is included in the range.

*index-num*:
*const-int-expr*

In the future, **end** will be added to index-num and indices will be able to include spans.
*const-int-expr*:

*Int*
*Id*
( *const-int-expr* )
*const-int-expr* + *const-int-expr*
*const-int-expr* - *const-int-expr*
*const-int-expr* * *const-int-expr*
- *const-int-expr* Checked with §B.14 test-flip5c.
*size-macro* §3.6.2 Checked with §B.13 test-flip5b.

Index expressions are a subset of regular expressions which enforces that indices are only integers[1]. *Id* in this case must be an index defined by a for macro.

The operator . and subscripting group left to right.

### 3.4.2 Convolution operator

*expression* ** *kernel-call*

The binary operator ** indicates convolution. The expression to the left must evaluate to a matrix of fixed size. On the right, a kernel function is referenced and a matrix is generated that matches the dimension of the expression on the left. A convolution performs a pointwise multiplication on the matrices and sums the elements of the resulting matrix [2].

### 3.4.3 Unary operator

- *expression*

The unary operator - negates the expression and has the same type. If the expression is a matrix, then every element is negated [3].

### 3.4.4 Exponential operator

*expression-1* ^ *expression-2*

The binary operator ^ indicates expression-1 being raised to the power of expression-2 [4]. expression-1 must be a float, an int, or a matrix. If it is a matrix, then each element in expression-1 is raised to the power of expression-2. expression-2 must be a float or an int. The result is either a float or a matrix of float.

### 3.4.5 Multiplicative operators

*expression-1* * *expression-2*

The binary operator * indicates pointwise multiplication. If both operands are matrices, then the element-by-element product is returned. In this case, both operands must have the equal dimensions. Otherwise, at least one of the expressions is a scalar. If either of the operands is a matrix, then the result is a matrix; else if either of the operands is a float, then the result is a float; otherwise both of the operands is an int and an int is returned.

*expression-1* / *expression-2*

The binary operator / indicates pointwise division. The same size considerations apply as for multiplication. If either operand is a matrix the result is a matrix [5]; otherwise the result is a float. Integer division does not exist

[1] Checked with §B.12 test-flip5a and §B.13 test-flip5b.
[2] Checked with §B.15 test-kernel1d.
[3] Checked with §B.23 test-matrixnegate.
[4] Checked with §B.31 test-scalarpowdiff.
[5] Checked with §B.22 test-matrixmatrix-div, §B.20 test-matrixfloat-div1 and §B.21 test-matrixfloat-div2.

in Synapse[1]. An expression that contains division must not be used when constant integers are required, as when defining the size of a matrix.

The results for division by zero are currently undefined, however, it may be defined in future versions of Synapse.

### 3.4.6 Additive operators

*expression-1* + *expression-2*

The binary operator * indicates pointwise addition[2]. The same size and type considerations apply as for multiplication.

*expression-1* **-** *expression-2*

The binary operator **-** indicates pointwise subtraction. The same size and type considerations apply as for multiplication.

## 3.5 Function Calls

### 3.5.1 Built-in functions

**sin**

**sin** ( x ) calculates the sine of $x$ in radians.

**cos**

**cos** ( x ) calculates the cosine of $x$ in radians.

**exp**

**exp** ( x ) calculates $e^x$.

### 3.5.2 User-defined functions

Activation functions may be called anywhere 3.4 can be used.
*activation-call*:
  *Id* ( )
  *Id* ( *fparam-list* ) §3.3.2

Kernel functions may be referenced directly after a convolution operator.
*kernel-call*:
  *Id* ( )
  *Id* ( *fparam-list* ) §3.3.2

The usefulness of kernel functions will be limited until spans are allowed in index expressions.

## 3.6 Macros

### 3.6.1 for macro

The for-macro makes it easier to connect a large number of modules, matrices of modules, and matrices of neurons.

[1] Checked with §B.37 fail-int-div.
[2] Checked with §B.16 test-matrixadd, §B.17 test-matrixadd2 and §B.18 test-matrixadd3.

*for-list*:
  *for-expression*
  *for-expression for-list*

*for-expression*:
  *Id* = [ *index-expression* ]

*span*:
  *index-num-1* **:** *index-num-2*  §3.4.1
  *index-num-1* **:** *const-int-expr* **:** *index-num-2*

Synaptic connections that use the for-macro will be evaluated for every combination of values in the Ids ranges[1]. The for-variable must appear in both the source and destination of the synaptic connection [2].

Used as an index to a module or neuron, it stands the smallest (ie. 1) and largest number that are well defined in that module or neuron respectively.

### 3.6.2 size macro

The size macro returns the size of a module or macro in the specified dimension.

*size-macro*:
  **size (** *indexable-expression* **,** *Int* **)**  §3.4.1

This macro can currently only be used when specifying indices[3] and cannot be used in spans (including in for-loops).

## 3.7 Scope

The scope of neurons are local to the current module. The neurons may be specified in any order. Neurons within or between modules may have circular or recurrent connections.

When connecting input and output neurons between modules, the module for which the neuron belongs must be specified.
*neuron-scoped*:
  *Id-1* **.** *Id-2*  §3.4.1

*Id-1* specifies the module and *Id-2* specifies the neuron contained in the module *Id-1*. While modules may be nested, only local neurons may be input or output neurons. Hence, only a single module is ever needed to reference a neuron.

All functions and module definitions have global scope. An activation function may only reference functions defined before it[4]. Activation functions may not be defined recursively. Neither function definitions may contain references to neurons.

Synaptic connections can connect modules and neurons regardless of location.

## 3.8 Concurrency

Unlike traditional programming languages, all of the values at the synaptic connections are calculated concurrently. Each synapse connection takes a single time step and at time $t$ only the values from time $t-1$ are used for the calculations.

The order of execution at each time step is[5]:

[1] Checked with §B.7 test-copy5a,§B.8 test-copymat and §B.9 test-copymat2.
[2] Checked with §B.38 fail-for-1.
[3] Checked with §B.13 test-flip5b.
[4] Checked with §B.4 test-afun-chain and §B.35 fail-afun-order.
[5] Checked with essentially every test case, but in particular §B.33 test-temporaloffset.

1. Input parameters are updated.
2. Neurons are updated using their corresponding synaptic connection.
3. Output parameter values are written.

Since it takes a while for the values from the input to be propagated throughout the program, the initial values are to be defined by the compiler runtime options. The compiler or interpreter must support the option of initialization of the neurons to zero. Other options, such as initialization by use of random distributions may also be supported. When the program starts writing output is also compiler or interpreter defined. It must at minimum support the option to start writing output as soon as it starts running, which means the initial output will be garbage.

While various inputs are supported, sequences of images or videos are well suited for reading and writing a large number of values.

## 3.9 Future Additions

The following additions are planned for Synapse.

The support for the size macro will be expanded. The keyword 'end' can be used in spans.

At any time $t+1$, the current version of Synapse only allows values from time $t$ to be referenced. Future versions will allow any $t$ or older neuron values to be referenced. This will be support by making $t$ a keyword that can be used in arrays. For example, $x$ would be the same thing as writing $x[t]$, $y[t, 1:10]$ would be the same thing as $y[1:10]$, and $x[t-1]$ and $y[t-2, 1:10]$ would refer to previous versions.

Some form of inline switch statements will be allowed in functions.

A dimension macro will be added that could be referenced in weight definitions. A way of automatically normalizing the dynamic kernels will be added. For example, Z may become a macro that stands for the sum of the weights of the current kernel.

Matrix constants can be defined in the form of: [[ 1, 2, 3, 4; 5, 6, 7, 8 ]].

Support for spike trains will be added by adding support for booleans and by adding support for Poisson spike generators.

# Chapter 4

# Project Plan

## 4.1 Process

In order to allow the most agility, I followed the "Release Early. Release Often" paradigm. Before implementing any functionality, I would first create a test case that would test it. After implementing the functionality and making sure that the test case works, I would commit both the changes and corresponding test case. My first goal, after defining an initial version of Language Reference Manual, was to enable the compilation of a very simple program. The first step, I thought, was to be able to read and write images. However, I ran into trouble getting the CamlImages project to compile and run and failed on all of the platforms that I tried (Windows, Cygwin, and Linux).

## 4.2 Style Guide and Naming Conventions

I used VIM to program in Ocaml and the native setup interfered with my development. I resorted to installing the extension OMLet created by David Baelde and using its style. It uses two spaces for indentation. The only behavior I disliked was when declaring mutually recursive functions it would indent the "and" way too much. In this case, the indentation should be removed or reduced.

There are some naming conventions that I have used. Indices (which indicates which element in a matrix is being referenced) should be distinquished from dimensions (the declared size of the matrix). The name "ind" is used for a single index and "indl" is used to reference a list of indices. Likewise, "dim" and "diml" reference a dimension and a list of dimensions.

Functions and variables should be all lower case with "underlines" being used for readability. Functions that produce strings should begin "string_of_". Functions that reduce the ambiguity of the type and dimensionality of expressions should begin with "resolve_".

An expression's type and dimensionality should not be resolved until every contained sub-expression is resolved.

## 4.3 Project Timeline

| | | | |
|---|---|---|---|
| March | Sun | 1 | Signed up to googlecode for SVN repository. |
| | Wed | 4 | First compiling version of parser, scanner, and abstract syntax tree. Not all functionality implemented. |
| | Sat | 7 | Started writing up the language reference manual. |
| | Tues | 10 | Submitted the language reference manual. |
| | Tues | 24 | Discovered the need for the sizes of the input parameters to be specified and implemented it. |
| | Fri | 27 | Added syntatically checked AST (sast.mli - although later renamed sast.ml). |
| April | Thurs | 9 | **MILESTONE** first version that compiles a program. B.30 test-scalarcopy works. |
| | Fri | 10 | B.26 test-scalaraddf works. |
| | Fri | 10 | B.27 test-scalaraddi works. |
| | Fri | 10 | B.28 test-scalararithmetic1 works. |
| | Fri | 10 | B.29 test-scalararithmetic2 works. |
| | Sat | 11 | B.7 test-copy5a works. |
| | Sat | 11 | B.12 test-flip5a works and made corresponding change in manual that was required to get this example to work. |
| | Sat | 11 | B.13 test-flip5b works, which uses **size** on input. |
| | Sat | 18 | B.8 test-copymat works. |
| | Sat | 18 | B.9 test-copymat2 works. |
| | Sat | 18 | B.1 test-afun1 works - tests activation function without any optional function parameters. |
| | Sat | 18 | B.2 test-afun2 works - tests activation function with optional parameter being overwritten. |
| | Sat | 18 | B.3 test-afun3 works - tests activation function where default value of optional parameter is used. |
| | Sat | 18 | B.9 test-copymat2 works. |
| | Sat | 18 | B.9 test-copymat2 works. |
| | Fri | 24 | B.25 test-module1a works - first working module definition test case. |
| | Fri | 24 | B.40 fail-module1a1 works. |
| | Sat | 25 | B.24 test-module1 added that tests multidimensional neurons, although there was a bug that I later found and fixed. |
| May | Sat | 2 | defined the timing of Synapse more clearly and modified the outputs of almost all of the test cases. |
| | Sat | 2 | stopped trying to fix the functionality to read and write PPM images so that I could focus on core functionality. |
| | Mon | 4 | B.43 fail-timing works. |
| | Mon | 4 | B.33 test-temporaloffset works. |
| | Thurs | 7 | B.16 test-matrixadd works. |
| | Thurs | 7 | B.17 test-matrixadd2 and B.24 test-module1 works. |
| | Fri | 8 | Freeze on adding functionality to focus on project report and testing. |

## 4.4 Development Environment

I used VIM on a Windows CYGWIN environment since Windows is installed on my laptop. I installed both the Windows and CYGWIN version of OCaml and later deleted the Windows version. There was a bug in the Windows interface which didn't appear in the console mode. At one point I tried to install CamlImages, which brought on a lot of headaches and no success. After this I tried to get everything to work without any additional packages.

I used Subversion through googlecode.com for source control and tracking, since Subversion is emperically better than CVS and anyone who continues to use CVS is stuck living in the dark ages. The main improvement, in my opinion, is the ability to rename and move files while keeping the file history. I used the Subversion client

TortoiseSVN, one of the few redeeming features of Windows.

## 4.5 Project Log

```
Revision: 1
Author:
Date: 5:09:30 PM, Sunday, March 01, 2009
Message:
Initial directory structure.
----
Added : /trunk
Added : /branches
Added : /tags

Revision: 2
Author: jonwilliford
Date: 5:23:19 PM, Sunday, March 01, 2009
Message:
Initial commit of a very simple language.
----
Added : /trunk/ast.mli
Added : /trunk/parser.mly

Revision: 3
Author: jonwilliford
Date: 11:17:00 AM, Monday, March 02, 2009
Message:
This version doesn't compile... just moving stuff around and adding files.
----
Added : /trunk/src
Added : /trunk/src/synap.ml
Added : /trunk/src/Makefile
Added : /trunk/src/ast.mli (Copy from path: /trunk/ast.mli, Revision, 2)
Added : /trunk/src/parser.mly (Copy from path: /trunk/parser.mly, Revision, 2)
Added : /trunk/src/scanner.mll
Deleted : /trunk/ast.mli
Deleted : /trunk/parser.mly
Added : /trunk/tests

Revision: 4
Author: jonwilliford
Date: 10:58:56 PM, Monday, March 02, 2009
Message:
Only synap.ml doesn't build.
----
Modified : /trunk/src/synap.ml
Modified : /trunk/src/ast.mli
Modified : /trunk/src/parser.mly
Modified : /trunk/src/scanner.mll

Revision: 7
Author: jonwilliford
Date: 10:49:58 PM, Tuesday, March 03, 2009
```

```
Message:
Made significant strives towards changing the parser such that it represents
Synapse. Currently doesn't compile.
----
Modified : /trunk/src/ast.mli
Modified : /trunk/src/parser.mly
Modified : /trunk/src/scanner.mll

Revision: 8
Author: jonwilliford
Date: 10:52:04 PM, Wednesday, March 04, 2009
Message:
Everything in scanning & parsing phase seems to compile.
----
Modified : /trunk/src/ast.mli
Modified : /trunk/src/parser.mly
Modified : /trunk/src/scanner.mll

Revision: 9
Author: jonwilliford
Date: 11:41:31 PM, Thursday, March 05, 2009
Message:
Compiles and implements more functionality. No conflicts. Kernel keyword
introduced. Eliminated brackets for optional parameters. begin, end, for, &
t not yet used.
----
Modified : /trunk/src/Makefile
Modified : /trunk/src/ast.mli
Modified : /trunk/src/parser.mly
Modified : /trunk/src/scanner.mll

Revision: 10
Author: jonwilliford
Date: 6:18:25 PM, Saturday, March 07, 2009
Message:
Another commit that compiles. for, end, indexing, synaptic connections,
scoped names added.
----
Modified : /trunk/src/ast.mli
Modified : /trunk/src/parser.mly
Modified : /trunk/src/scanner.mll
Added : /trunk/docs
Added : /trunk/docs/Language Reference Manual.odt

Revision: 11
Author: jonwilliford
Date: 6:20:16 PM, Saturday, March 07, 2009
Message:
Added LaTeX and PDF versions of the language reference manual.
----
Added : /trunk/docs/Synapse Language Refence Manual.pdf
Added : /trunk/docs/Synapse Language Refence Manual.tex

Revision: 12
```

```
Author: jonwilliford
Date: 11:28:03 PM, Saturday, March 07, 2009
Message:
Updated LRM and made some corresponding minor changes in source.
----
Modified : /trunk/src
Modified : /trunk/src/ast.mli
Modified : /trunk/src/parser.mly
Modified : /trunk/docs
Modified : /trunk/docs/Synapse Language Refence Manual.pdf
Modified : /trunk/docs/Synapse Language Refence Manual.tex
Modified : /trunk/docs/Language Reference Manual.odt

Revision: 13
Author: jonwilliford
Date: 6:50:12 PM, Sunday, March 08, 2009
Message:
Minor changes to code.  More significant changes to documentation.
----
Modified : /trunk/src/Makefile
Modified : /trunk/src/ast.mli
Modified : /trunk/src/parser.mly
Modified : /trunk/docs/Synapse Language Refence Manual.pdf
Modified : /trunk/docs/Synapse Language Refence Manual.tex

Revision: 14
Author: jonwilliford
Date: 8:22:49 PM, Monday, March 09, 2009
Message:
Minor changes?
----
Modified : /trunk/src/Makefile
Modified : /trunk/src/parser.mly
Modified : /trunk/docs/Synapse Language Refence Manual.pdf
Modified : /trunk/docs/Synapse Language Refence Manual.tex

Revision: 15
Author: jonwilliford
Date: 8:40:00 PM, Tuesday, March 10, 2009
Message:
Made significant changes to LRM.  Deleted old LRM.
----
Modified : /trunk/docs/Synapse Language Refence Manual.pdf
Modified : /trunk/docs/Synapse Language Refence Manual.tex
Deleted : /trunk/docs/Language Reference Manual.odt

Revision: 16
Author: jonwilliford
Date: 10:18:53 PM, Tuesday, March 10, 2009
Message:
Version submitting to COMS 4115.
----
Modified : /trunk/docs/Synapse Language Refence Manual.pdf
```

```
Modified : /trunk/docs/Synapse Language Refence Manual.tex

Revision: 17
Author: jonwilliford
Date: 8:02:29 PM, Wednesday, March 18, 2009
Message:
Added support for pragma().  Modified Makefile to allow Str module.
----
Modified : /trunk/src/Makefile
Modified : /trunk/src/ast.mli
Modified : /trunk/src/parser.mly
Modified : /trunk/src/scanner.mll
Added : /trunk/src/testall.sh

Revision: 18
Author: jonwilliford
Date: 8:04:10 PM, Wednesday, March 18, 2009
Message:
Tried something... going a different direction now.
----
Replacing : /trunk/src/synap.ml

Revision: 19
Author: jonwilliford
Date: 9:44:03 PM, Thursday, March 19, 2009
Message:
Fixed an error in the section on scoping.  The rules previously didn't allow
indices to be specified on referenced modules.
----
Modified : /trunk/docs/Synapse Language Refence Manual.pdf
Modified : /trunk/docs/Synapse Language Refence Manual.tex

Revision: 20
Author: jonwilliford
Date: 7:31:19 PM, Sunday, March 22, 2009
Message:
Now finds the inputs and outputs of the program and prints them out.
----
Modified : /trunk/src/synap.ml
Modified : /trunk/src/ast.mli
Modified : /trunk/src/parser.mly
Modified : /trunk/src/scanner.mll

Revision: 21
Author: jonwilliford
Date: 10:27:56 PM, Tuesday, March 24, 2009
Message:
Added "input" keyword for declaring input parameters.
----
Modified : /trunk/docs/Synapse Language Refence Manual.tex

Revision: 22
Author: jonwilliford
```

```
Date: 10:48:31 PM, Tuesday, March 24, 2009
Message:
Aesthetic changes.  Plus modified program definition for the input.
----
Modified : /trunk/docs/Synapse Language Refence Manual.pdf
Modified : /trunk/docs/Synapse Language Refence Manual.tex

Revision: 23
Author: jonwilliford
Date: 10:50:18 PM, Tuesday, March 24, 2009
Message:
Dimensions and indices are represented as one.  They will be check during
semantic analysis.  Made some aesthetic changes.
----
Modified : /trunk/src/ast.mli
Modified : /trunk/src/parser.mly
Modified : /trunk/src/scanner.mll

Revision: 24
Author: jonwilliford
Date: 12:19:31 PM, Friday, March 27, 2009
Message:

----
Modified : /trunk/src/Makefile

Revision: 25
Author: jonwilliford
Date: 1:18:14 PM, Friday, March 27, 2009
Message:
Adding syntactically checked AST file.
----
Added : /trunk/src/sast.mli

Revision: 26
Author: jonwilliford
Date: 11:15:43 PM, Friday, March 27, 2009
Message:
Trying to get a static semantic checking to work on a simple example.
----
Modified : /trunk/src/Makefile
Added : /trunk/src/params.ml
Added : /trunk/src/types.mli
Added : /trunk/src/translate.ml
Modified : /trunk/src/sast.mli

Revision: 27
Author: jonwilliford
Date: 12:24:52 PM, Saturday, March 28, 2009
Message:
Actually compiles ...
----
Modified : /trunk/src/Makefile
```

```
Modified : /trunk/src/translate.ml

Revision: 28
Author: jonwilliford
Date: 3:58:56 PM, Saturday, March 28, 2009
Message:
Modified the format of the for-macro.  Complies and prints out the input and
output parameters.
----
Modified : /trunk/src/Makefile
Modified : /trunk/src/parser.mly
Modified : /trunk/docs/Synapse Language Refence Manual.pdf
Modified : /trunk/docs/Synapse Language Refence Manual.tex
Modified : /trunk/src/params.ml
Modified : /trunk/src/translate.ml

Revision: 29
Author: jonwilliford
Date: 3:59:35 PM, Saturday, March 28, 2009
Message:

----
Added : /trunk/tests/test-copy5.syn
Added : /trunk/tests/test-const1.syn

Revision: 30
Author: jonwilliford
Date: 9:41:31 PM, Saturday, March 28, 2009
Message:
Another version that compiles and prints out the parameters...
----
Modified : /trunk/src/params.ml
Modified : /trunk/src/translate.ml
Modified : /trunk/src/sast.mli

Revision: 31
Author: jonwilliford
Date: 8:27:12 PM, Monday, March 30, 2009
Message:
Doesn't compile.
----
Modified : /trunk/src/Makefile
Added : /trunk/src/sast.ml (Copy from path: /trunk/src/sast.mli, Revision, 30)
Modified : /trunk/src/params.ml
Modified : /trunk/src/translate.ml
Deleted : /trunk/src/sast.mli

Revision: 32
Author: jonwilliford
Date: 7:27:09 PM, Tuesday, March 31, 2009
Message:
Compiles.
----
```

```
Modified : /trunk/src/Makefile
Modified : /trunk/src/params.ml
Modified : /trunk/src/translate.ml

Revision: 33
Author: jonwilliford
Date: 9:43:24 PM, Tuesday, March 31, 2009
Message:
Sets the size of the input from the input declaration.
----
Modified : /trunk/src/sast.ml
Modified : /trunk/src/params.ml
Modified : /trunk/src/types.mli
Modified : /trunk/src/translate.ml

Revision: 34
Author: jonwilliford
Date: 9:49:25 PM, Tuesday, March 31, 2009
Message:
Rearranged some code.
----
Modified : /trunk/src/params.ml
Modified : /trunk/src/translate.ml

Revision: 35
Author: jonwilliford
Date: 10:49:20 PM, Wednesday, April 01, 2009
Message:

----
Modified : /trunk/src/synap.ml
Modified : /trunk/src/Makefile
Modified : /trunk/src/testall.sh
Modified : /trunk/src/sast.ml
Modified : /trunk/src/params.ml
Modified : /trunk/src/translate.ml

Revision: 36
Author: jonwilliford
Date: 11:50:54 PM, Thursday, April 02, 2009
Message:
I didn't make a lot of changes, but I didn't break anything! ... I don't think
...
----
Modified : /trunk/src/synap.ml
Modified : /trunk/src/sast.ml
Added : /trunk/tests/in5.txt
Modified : /trunk/src/params.ml
Added : /trunk/src/testone.sh
Added : /trunk/tests/test-copy5.args

Revision: 37
Author: jonwilliford
```

```
Date: 6:10:14 PM, Saturday, April 04, 2009
Message:
translate.ml is being renamed to translate1.ml and translate2.ml (now empty)
has been added.  Code compiles and runs, even though it doesn't do anything
useful.
----
Modified : /trunk/src/synap.ml
Modified : /trunk/src/Makefile
Modified : /trunk/src/ast.mli
Modified : /trunk/src/sast.ml
Added : /trunk/src/translate1.ml (Copy from path: /trunk/src/translate.ml,
Revision, 35)
Added : /trunk/src/translate2.ml
Modified : /trunk/src/params.ml
Added : /trunk/src/validate.ml
Modified : /trunk/src/testone.sh
Deleted : /trunk/src/translate.ml

Revision: 38
Author: jonwilliford
Date: 6:35:02 PM, Saturday, April 04, 2009
Message:
Fixed bug.
----
Modified : /trunk/src/translate1.ml
Modified : /trunk/src/params.ml
Modified : /trunk/src/testone.sh

Revision: 39
Author: jonwilliford
Date: 10:36:59 PM, Saturday, April 04, 2009
Message:
Minor fixes to manual.  Compiles and throws exceptions on run.
----
Modified : /trunk/docs/Synapse Language Refence Manual.pdf
Modified : /trunk/docs/Synapse Language Refence Manual.tex
Modified : /trunk/src/sast.ml
Modified : /trunk/src/translate1.ml
Modified : /trunk/src/translate2.ml

Revision: 40
Author: jonwilliford
Date: 12:08:50 AM, Sunday, April 05, 2009
Message:
Adding simpler test case.
----
Added : /trunk/tests/test-scalarcopy.out
Modified : /trunk/src/testone.sh
Added : /trunk/tests/test-scalarcopy.args
Added : /trunk/tests/fibdec.txt
Added : /trunk/tests/test-scalarcopy.syn

Revision: 41
```

```
Author: jonwilliford
Date: 4:04:44 PM, Sunday, April 05, 2009
Message:
Adding printer for Sast.
----
Modified : /trunk/src/synap.ml
Modified : /trunk/src/Makefile
Added : /trunk/src/printer.ml
Modified : /trunk/src/sast.ml
Modified : /trunk/src/translate1.ml
Modified : /trunk/src/params.ml
Modified : /trunk/src/validate.ml

Revision: 42
Author: jonwilliford
Date: 7:48:55 PM, Sunday, April 05, 2009
Message:
Always getting closer...  it looks like in order to get the simple test case
working, I just need to create an array for the parameters to store their
values and then write to the code to read and write the parameters.  This
version does compile.
----
Modified : /trunk/src/Makefile
Modified : /trunk/src/printer.ml
Modified : /trunk/src/sast.ml
Modified : /trunk/src/translate2.ml
Modified : /trunk/src/types.mli

Revision: 43
Author: jonwilliford
Date: 9:14:11 PM, Wednesday, April 08, 2009
Message:
Very close to having simple case.  "Just" need to eval synapses.
----
Modified : /trunk/src/synap.ml
Modified : /trunk/src/sast.ml
Modified : /trunk/src/translate2.ml
Modified : /trunk/src/params.ml

Revision: 44
Author: jonwilliford
Date: 10:04:09 PM, Thursday, April 09, 2009
Message:
First working commit!  (for the very simplest case...)
----
Modified : /trunk/src/synap.ml
Modified : /trunk/tests/test-scalarcopy.out

Revision: 45
Author: jonwilliford
Date: 11:08:51 AM, Friday, April 10, 2009
Message:
Testall script now works.
```

```
----
Modified : /trunk/src/testall.sh
Modified : /trunk/src/testone.sh
Modified : /trunk/tests/test-scalarcopy.args

Revision: 46
Author: jonwilliford
Date: 11:10:47 AM, Friday, April 10, 2009
Message:

----
Deleted : /trunk/tests/test-copy5.args
Deleted : /trunk/tests/test-copy5.syn
Deleted : /trunk/tests/test-const1.syn

Revision: 47
Author: jonwilliford
Date: 1:08:19 PM, Friday, April 10, 2009
Message:
Now performs addition.  Added test case to show that it works.
----
Modified : /trunk/src/synap.ml
Modified : /trunk/src/ast.mli
Modified : /trunk/src/scanner.mll
Modified : /trunk/src/testall.sh
Modified : /trunk/src/printer.ml
Modified : /trunk/src/sast.ml
Modified : /trunk/src/translate1.ml
Modified : /trunk/src/translate2.ml
Modified : /trunk/src/types.mli
Modified : /trunk/src/validate.ml
Added : /trunk/tests/test-scalaraddf.args
Added : /trunk/tests/test-scalaraddf.out
Added : /trunk/tests/test-scalaraddf.syn

Revision: 48
Author: jonwilliford
Date: 1:46:47 PM, Friday, April 10, 2009
Message:
Adding integers seems to work.  Corresponding test case added.
----
Modified : /trunk/src/synap.ml
Modified : /trunk/src/translate2.ml
Added : /trunk/tests/test-scalaraddi.args
Added : /trunk/tests/test-scalaraddi.out
Added : /trunk/tests/test-scalaraddi.syn

Revision: 49
Author: jonwilliford
Date: 3:59:46 PM, Friday, April 10, 2009
Message:
Fixed a bug and added a corresponding test case.
----
```

```
Modified : /trunk/src/synap.ml
Modified : /trunk/src/Makefile
Modified : /trunk/src/params.ml
Added : /trunk/tests/test-scalararithmetic1.args
Added : /trunk/tests/test-scalararithmetic1.out
Added : /trunk/tests/test-scalararithmetic1.syn

Revision: 50
Author: jonwilliford
Date: 4:18:48 PM, Friday, April 10, 2009
Message:
Added a test case that uses two input files and tests +,-,/, and *.
----
Modified : /trunk/src/printer.ml
Added : /trunk/tests/test-scalararithmetic2.out
Added : /trunk/tests/scalar1.txt
Added : /trunk/tests/test-scalararithmetic2.args
Added : /trunk/tests/test-scalararithmetic2.syn

Revision: 51
Author: jonwilliford
Date: 5:27:09 PM, Friday, April 10, 2009
Message:
Trying to get test-copy5.syn to work.  Still successfully runs the other tests
.
----
Modified : /trunk/src/sast.ml
Modified : /trunk/src/translate1.ml
Modified : /trunk/src/types.mli
Modified : /trunk/src/validate.ml
Added : /trunk/tests/test-copy5.args
Added : /trunk/tests/test-copy5.syn

Revision: 52
Author: jonwilliford
Date: 7:41:39 PM, Saturday, April 11, 2009
Message:
Very close to get test-copy5a.syn to work!  Currently inverts the results.
----
Modified : /trunk/src/synap.ml
Modified : /trunk/src/Makefile
Added : /trunk/src/eval.ml
Modified : /trunk/src/printer.ml
Modified : /trunk/src/sast.ml
Modified : /trunk/src/translate1.ml
Modified : /trunk/src/translate2.ml
Modified : /trunk/tests/in5.txt
Added : /trunk/tests/test-copy5a.args
Added : /trunk/tests/test-copy5a.syn

Revision: 53
Author: jonwilliford
Date: 9:10:54 PM, Saturday, April 11, 2009
```

```
Message:
test-copy5a.syn now works!
----
Modified : /trunk/src/synap.ml
Modified : /trunk/tests/in5.txt
Added : /trunk/tests/test-copy5a.out

Revision: 54
Author: jonwilliford
Date: 9:15:39 PM, Saturday, April 11, 2009
Message:
Added test case that currently fails.
----
Added : /trunk/tests/test-flip5a.syn
Added : /trunk/tests/test-flip5a.args

Revision: 55
Author: jonwilliford
Date: 9:40:28 PM, Saturday, April 11, 2009
Message:
test-flip5a.syn now works.  Had to make corresponding change in manual (it
didn't allow [x+1] in index.
----
Modified : /trunk/src/synap.ml
Modified : /trunk/src/parser.mly
Modified : /trunk/docs/Synapse Language Refence Manual.pdf
Modified : /trunk/docs/Synapse Language Refence Manual.tex
Modified : /trunk/src/translate2.ml
Added : /trunk/tests/test-flip5a.out
Modified : /trunk/tests/test-flip5a.syn

Revision: 56
Author: jonwilliford
Date: 10:31:12 PM, Saturday, April 11, 2009
Message:
test-flip5b.syn now works.  It uses size() on input parameter.
----
Modified : /trunk/src/parser.mly
Modified : /trunk/docs/Synapse Language Refence Manual.tex
Modified : /trunk/src/eval.ml
Modified : /trunk/src/sast.ml
Modified : /trunk/src/translate1.ml
Modified : /trunk/src/translate2.ml
Modified : /trunk/src/validate.ml
Added : /trunk/tests/test-flip5b.out
Added : /trunk/tests/test-flip5b.args
Added : /trunk/tests/test-flip5b.syn

Revision: 57
Author: jonwilliford
Date: 8:42:56 PM, Friday, April 17, 2009
Message:
Got 2D matrices to work and added corresponding test case (test-copymat.syn).
```

```
----
Modified : /trunk/src/printer.ml
Modified : /trunk/src/translate1.ml
Added : /trunk/tests/test-copymat.out
Added : /trunk/tests/test-copymat.args
Added : /trunk/tests/test-copymat.syn

Revision: 58
Author: jonwilliford
Date: 12:15:15 PM, Saturday, April 18, 2009
Message:
Okay, I lied. test-copymat.syn wasn't working, but it is now.  Additionally
test-copymat2.syn works.
----
Modified : /trunk/src/synap.ml
Modified : /trunk/src/eval.ml
Modified : /trunk/tests/test-copymat.out
Added : /trunk/tests/test-copymat2.out
Added : /trunk/tests/in10.txt
Added : /trunk/tests/in16.txt
Added : /trunk/tests/test-copymat2.args
Added : /trunk/tests/test-copymat2.syn

Revision: 59
Author: jonwilliford
Date: 5:39:41 PM, Saturday, April 18, 2009
Message:
Activation functions without any optional parameters now work.  Added test
case test-afun1.syn to test this functionality.
----
Modified : /trunk/src/synap.ml
Modified : /trunk/src/ast.mli
Modified : /trunk/src/parser.mly
Modified : /trunk/src/eval.ml
Modified : /trunk/src/printer.ml
Modified : /trunk/src/sast.ml
Modified : /trunk/src/translate1.ml
Modified : /trunk/src/translate2.ml
Modified : /trunk/src/validate.ml
Added : /trunk/tests/test-afun1.out
Added : /trunk/tests/test-afun1.args
Added : /trunk/tests/test-afun1.syn

Revision: 60
Author: jonwilliford
Date: 7:44:40 PM, Saturday, April 18, 2009
Message:
Added the ability for optional parameters of functions to be specified with
int (that are immediately cast as floats).  Added corresponding test case test
-afun2.syn.
----
Modified : /trunk/src/parser.mly
Added : /trunk/tests/test-afun2.out
Added : /trunk/tests/test-afun2.args
```

```
Added : /trunk/tests/test-afun2.syn

Revision: 61
Author: jonwilliford
Date: 7:59:08 PM, Saturday, April 18, 2009
Message:
Previously the compiler would crash if the the default function parameter wasn
'''t overwritten.  Fixed this issue and added corresponding the corresponding
test case test-afun3.syn.
----
Modified : /trunk/src/eval.ml
Added : /trunk/tests/test-afun3.out
Added : /trunk/tests/test-afun3.args
Added : /trunk/tests/test-afun3.syn

Revision: 62
Author: jonwilliford
Date: 9:28:52 PM, Saturday, April 18, 2009
Message:
Fixed mistake: the neuron-list indicating input and output neurons in the
modules needed to allow dimensions.
----
Modified : /trunk/docs/Synapse Language Refence Manual.pdf
Modified : /trunk/docs/Synapse Language Refence Manual.tex

Revision: 63
Author: jonwilliford
Date: 11:04:02 PM, Saturday, April 18, 2009
Message:
Previous definitions didn't actually let program parameters appear as the
destination of the synaptic connection.
----
Modified : /trunk/docs/Synapse Language Refence Manual.pdf
Modified : /trunk/docs/Synapse Language Refence Manual.tex

Revision: 64
Author: jonwilliford
Date: 7:53:22 PM, Monday, April 20, 2009
Message:
Didn't break anything.  Working to get modules definitions to work.
----
Modified : /trunk/src/ast.mli
Modified : /trunk/src/parser.mly
Modified : /trunk/src/eval.ml
Modified : /trunk/src/printer.ml
Modified : /trunk/src/sast.ml
Modified : /trunk/src/translate1.ml
Modified : /trunk/src/translate2.ml
Modified : /trunk/src/params.ml
Modified : /trunk/src/validate.ml

Revision: 65
Author: jonwilliford
Date: 9:06:38 PM, Monday, April 20, 2009
```

```
Message:
Still didn't break anything.
----
Modified : /trunk/src/ast.mli
Modified : /trunk/src/parser.mly
Modified : /trunk/src/printer.ml
Modified : /trunk/src/sast.ml
Modified : /trunk/src/translate1.ml
Modified : /trunk/src/params.ml
Modified : /trunk/src/validate.ml

Revision: 66
Author: jonwilliford
Date: 9:06:59 PM, Tuesday, April 21, 2009
Message:
Haven't broken anything, but I'm close... to getting simple module test case
to work.
----
Modified : /trunk/src/synap.ml
Modified : /trunk/src/eval.ml
Modified : /trunk/src/printer.ml
Modified : /trunk/src/sast.ml
Modified : /trunk/src/translate1.ml
Modified : /trunk/src/translate2.ml
Modified : /trunk/src/validate.ml

Revision: 67
Author: jonwilliford
Date: 6:56:23 PM, Thursday, April 23, 2009
Message:
Getting a lot closer to getting the simple module case to work.  translate2.ml
 needs to check if an Id is actually a local neuron reference.
----
Modified : /trunk/src/synap.ml
Modified : /trunk/src/printer.ml
Modified : /trunk/src/sast.ml
Modified : /trunk/src/translate1.ml
Modified : /trunk/src/validate.ml

Revision: 68
Author: jonwilliford
Date: 1:19:49 PM, Friday, April 24, 2009
Message:
A simple module test case, test-module1a.syn now works.
----
Modified : /trunk/src/eval.ml
Modified : /trunk/src/printer.ml
Modified : /trunk/src/sast.ml
Modified : /trunk/src/translate1.ml
Modified : /trunk/src/validate.ml
Added : /trunk/tests/test-module1a.out
Added : /trunk/tests/test-module1a.args
Added : /trunk/tests/test-module1a.syn
```

```
Revision: 69
Author: jonwilliford
Date: 2:54:08 PM, Friday, April 24, 2009
Message:
Added CheckFail() function in testall.sh script.  Simply returns "SUCCESS"
when error is thrown.  Should eventually check the error returned.
----
Modified : /trunk/src/testall.sh
Added : /trunk/tests/fail-fail.args
Added : /trunk/tests/fail-fail.syn

Revision: 70
Author: jonwilliford
Date: 3:57:27 PM, Friday, April 24, 2009
Message:
Added checks to make sure that external neuron references do not reference
inter-neurons and that the other types (inputs,output) types are referenced
correctly.  Added tests that insure that input and output neurons are used
correctly.
----
Modified : /trunk/src/sast.ml
Modified : /trunk/src/translate1.ml
Added : /trunk/tests/fail-module1a1.args
Added : /trunk/tests/fail-module1a1.syn
Added : /trunk/tests/fail-module1a2.args
Added : /trunk/tests/fail-module1a2.syn

Revision: 71
Author: jonwilliford
Date: 6:26:52 PM, Friday, April 24, 2009
Message:
Getting close to allowing neurons to contain matrices.
----
Modified : /trunk/src/synap.ml
Modified : /trunk/src/eval.ml
Modified : /trunk/src/sast.ml
Modified : /trunk/src/translate1.ml
Modified : /trunk/src/translate2.ml
Modified : /trunk/src/validate.ml

Revision: 72
Author: jonwilliford
Date: 10:30:08 AM, Saturday, April 25, 2009
Message:
Implemented the ability for neurons to be multidimensional matrices.  Added
the test case test-module1.syn for this functionality.
----
Modified : /trunk/src/synap.ml
Modified : /trunk/src/printer.ml
Modified : /trunk/src/translate1.ml
Modified : /trunk/src/validate.ml
Added : /trunk/tests/test-module1.args
Added : /trunk/tests/test-module1.out
```

```
Added : /trunk/tests/test-module1.syn

Revision: 73
Author: jonwilliford
Date: 8:43:16 PM, Wednesday, April 29, 2009
Message:
Getting somewhat close to being able to read in PPM files.
----
Modified : /trunk/src/synap.ml
Modified : /trunk/src/Makefile
Modified : /trunk/src/sast.ml
Modified : /trunk/src/params.ml
Added : /trunk/src/ppm.ml
Added : /trunk/tests/images
Added : /trunk/tests/images/barbara.ppm

Revision: 74
Author: jonwilliford
Date: 11:49:43 AM, Saturday, May 02, 2009
Message:
Much closer to reading and writing images, but still some errors.  Other cases
 still work.
----
Modified : /trunk/src/synap.ml
Modified : /trunk/src/eval.ml
Modified : /trunk/src/sast.ml
Modified : /trunk/src/translate2.ml
Modified : /trunk/src/validate.ml
Modified : /trunk/src/ppm.ml

Revision: 75
Author: jonwilliford
Date: 5:50:17 PM, Sunday, May 03, 2009
Message:
The code in this commit is more exact in the order of execution.  During each
time step the following order is followed:
1. The input params are read in.
2. The synapses calculate their temporary values and then updates the current
values with these temporary values.
3. The output params are written.

There are some really strange errors that are occurring with images.  It seems
 that the code is randomly throwing extra pixels in the output image.  The
results are consistent between runs.
----
Modified : /trunk/src/synap.ml
Modified : /trunk/src/eval.ml
Modified : /trunk/src/printer.ml
Modified : /trunk/src/sast.ml
Modified : /trunk/src/translate1.ml
Modified : /trunk/src/translate2.ml
Modified : /trunk/src/params.ml
Modified : /trunk/tests/test-scalararithmetic1.out
Modified : /trunk/tests/test-scalaraddi.out
```

```
Modified : /trunk/src/validate.ml
Modified : /trunk/tests/test-scalaraddf.out
Modified : /trunk/tests/test-scalarcopy.out
Modified : /trunk/tests/test-module1.out
Modified : /trunk/src/ppm.ml
Modified : /trunk/tests/test-afun1.out
Modified : /trunk/tests/test-afun2.out
Modified : /trunk/tests/test-afun3.out
Modified : /trunk/tests/test-copy5a.out
Modified : /trunk/tests/test-copymat.out
Modified : /trunk/tests/test-copymat2.out
Modified : /trunk/tests/test-flip5a.out
Modified : /trunk/tests/test-flip5b.out
Modified : /trunk/tests/test-module1a.out
Modified : /trunk/tests/test-scalararithmetic2.out

Revision: 76
Author: jonwilliford
Date: 9:23:46 PM, Monday, May 04, 2009
Message:
Added two test cases and made the corresponding changes to make them run
successfully.  The test case fail-timing.syn should fail because one of the
neurons is not connected to any synapse.  The test case test-temporaloffset.
syn previously didn't work because the type of the synapse expression was not
propagated to the neuron.
----
Modified : /trunk/src/synap.ml
Modified : /trunk/src/testall.sh
Modified : /trunk/src/printer.ml
Modified : /trunk/src/translate2.ml
Modified : /trunk/src/validate.ml
Added : /trunk/tests/fail-timing.args
Added : /trunk/tests/fail-timing.syn
Added : /trunk/tests/test-temporaloffset.args
Added : /trunk/tests/test-temporaloffset.out
Added : /trunk/tests/test-temporaloffset.syn

Revision: 77
Author: jonwilliford
Date: 12:01:36 PM, Thursday, May 07, 2009
Message:
Implemented matrix arithmetic and matrix synapse connections.  Added tests/
test-matrixadd.syn to test this functionality.
----
Modified : /trunk/src/synap.ml
Modified : /trunk/src/parser.mly
Modified : /trunk/src/eval.ml
Modified : /trunk/src/printer.ml
Modified : /trunk/src/sast.ml
Modified : /trunk/src/translate1.ml
Modified : /trunk/src/translate2.ml
Modified : /trunk/src/params.ml
Modified : /trunk/src/types.mli
```

```
Modified : /trunk/src/validate.ml
Added : /trunk/tests/test-matrixadd.args
Added : /trunk/tests/test-matrixadd.out
Added : /trunk/tests/test-matrixadd.syn

Revision: 78
Author: jonwilliford
Date: 1:12:04 PM, Thursday, May 07, 2009
Message:
Added debug information for the neuron (included the name of the neuron and
the parent module).
----
Modified : /trunk/src/printer.ml
Modified : /trunk/src/sast.ml
Modified : /trunk/src/translate1.ml
Modified : /trunk/src/translate2.ml

Revision: 79
Author: jonwilliford
Date: 9:18:58 PM, Thursday, May 07, 2009
Message:
Found a bug in the implementation and in test-module1.out; fixed both.  Added
test case test-matrixadd2.syn, which basically tests the same thing.
----
Modified : /trunk/src/synap.ml
Modified : /trunk/src/eval.ml
Modified : /trunk/src/printer.ml
Modified : /trunk/src/translate1.ml
Modified : /trunk/src/translate2.ml
Added : /trunk/tests/test-matrixadd2.out
Added : /trunk/tests/in5b.txt
Added : /trunk/tests/test-matrixadd2.args
Added : /trunk/tests/test-matrixadd2.syn
Modified : /trunk/tests/test-module1.args
Modified : /trunk/tests/test-module1.out

Revision: 80
Author: jonwilliford
Date: 10:23:24 PM, Thursday, May 07, 2009
Message:
Added functionality of matrix arithmetic within modules.  Added corresponding
test case test-matrixadd3.syn.  Corrected test-matrixadd2.out.
----
Modified : /trunk/src/printer.ml
Modified : /trunk/src/sast.ml
Modified : /trunk/src/translate1.ml
Modified : /trunk/src/translate2.ml
Modified : /trunk/tests/test-matrixadd2.out
Added : /trunk/tests/test-matrixadd3.args
Added : /trunk/tests/test-matrixadd3.out
Added : /trunk/tests/test-matrixadd3.syn

Revision: 81
Author: jonwilliford
```

```
Date: 11:15:34 AM, Friday, May 08, 2009
Message:
Added weight / kernel functions and added the corresponding test case test-
kernel2d.syn.
----
Modified : /trunk/src/synap.ml
Modified : /trunk/src/eval.ml
Modified : /trunk/src/sast.ml
Modified : /trunk/src/translate1.ml
Modified : /trunk/src/translate2.ml
Modified : /trunk/src/validate.ml
Added : /trunk/tests/test-kernel2d.args
Added : /trunk/tests/test-kernel2d.out
Added : /trunk/tests/test-kernel2d.syn

Revision: 82
Author: jonwilliford
Date: 12:36:13 PM, Friday, May 08, 2009
Message:
Added functionality for raising to the power (^).  Added corresponding test
case test-scalarpowdiff.syn.
----
Modified : /trunk/src/eval.ml
Added : /trunk/tests/test-scalarpowdiff.args
Added : /trunk/tests/test-scalarpowdiff.out
Added : /trunk/tests/test-scalarpowdiff.syn

Revision: 83
Author: jonwilliford
Date: 2:18:55 PM, Friday, May 08, 2009
Message:
Added code to handle negation and "uni-operators" (such as sin,cos,exp).
Added test cases that test negation and exp (but not yet sin and cos).
----
Modified : /trunk/src/ast.mli
Modified : /trunk/src/eval.ml
Modified : /trunk/src/parser.mly
Modified : /trunk/src/printer.ml
Modified : /trunk/src/sast.ml
Modified : /trunk/src/scanner.mll
Modified : /trunk/src/translate1.ml
Modified : /trunk/src/translate2.ml
Modified : /trunk/src/validate.ml
Added : /trunk/tests/test-matrixexp.args
Added : /trunk/tests/test-matrixexp.out
Added : /trunk/tests/test-matrixexp.syn
Added : /trunk/tests/test-matrixnegate.args
Added : /trunk/tests/test-matrixnegate.out
Added : /trunk/tests/test-matrixnegate.syn

Revision: 84
Author: jonwilliford
Date: 5:38:36 PM, Friday, May 08, 2009
Message:
```

```
Initial commit of the class project report.
----
Added : /trunk/docs/ProjectLog.txt
Added : /trunk/docs/Synapse Language Project Report.pdf
Added : /trunk/docs/Synapse Language Project Report.tex
Modified : /trunk/docs/Synapse Language Refence Manual.pdf
Modified : /trunk/docs/Synapse Language Refence Manual.tex
Added : /trunk/docs/Synapse_LRM_Chapter.tex
Added : /trunk/docs/Synapse_Report_Architectural_Design.tex
Added : /trunk/docs/Synapse_Report_Compiler_Source.tex
Added : /trunk/docs/Synapse_Report_Intro_Chapter.tex
Added : /trunk/docs/Synapse_Report_Language_Tutorial.tex
Added : /trunk/docs/Synapse_Report_Lessons_Learned.tex
Added : /trunk/docs/Synapse_Report_Project_Plan.tex
Added : /trunk/docs/Synapse_Report_Test_Plan.tex
Added : /trunk/docs/Synapse_Report_Tests.tex

Revision: 85
Author: jonwilliford
Date: 10:51:02 AM, Saturday, May 09, 2009
Message:
Removing the test that uses begin and end, since this functionality will not
be available at this release.
----
Modified : /trunk/src/ast.mli
Modified : /trunk/src/parser.mly
Modified : /trunk/src/printer.ml
Modified : /trunk/src/sast.ml
Modified : /trunk/src/scanner.mll
Modified : /trunk/src/translate1.ml
Modified : /trunk/src/validate.ml
Deleted : /trunk/tests/test-copy5.args
Deleted : /trunk/tests/test-copy5.syn

Revision: 86
Author: jonwilliford
Date: 12:06:57 PM, Saturday, May 09, 2009
Message:
Added support for multiplying matrices with scalars and added test cases that
test this (with multiplication on both sides) and the constants pi and e.
----
Modified : /trunk/src/translate2.ml
Added : /trunk/tests/test-constant-e.args
Added : /trunk/tests/test-constant-e.out
Added : /trunk/tests/test-constant-e.syn
Added : /trunk/tests/test-constant-pi.args
Added : /trunk/tests/test-constant-pi.out
Added : /trunk/tests/test-constant-pi.syn

Revision: 87
Author: jonwilliford
Date: 12:29:19 PM, Saturday, May 09, 2009
Message:
```

Made sure activation functions have arguments that are scalars with fail-afun-mat.syn. Also added some tests that I previously failed to added.
----
Modified : /trunk/src/translate1.ml
Modified : /trunk/src/translate2.ml
Modified : /trunk/src/validate.ml
Added : /trunk/tests/fail-afun-mat.args
Added : /trunk/tests/fail-afun-mat.syn
Added : /trunk/tests/fail-noinput.args
Added : /trunk/tests/fail-noinput.syn
Added : /trunk/tests/fail-reserved-word-end.args
Added : /trunk/tests/fail-reserved-word-end.syn
Added : /trunk/tests/fail-reserved-word-t.args
Added : /trunk/tests/fail-reserved-word-t.syn

Revision: 88
Author: jonwilliford
Date: 12:50:45 PM, Saturday, May 09, 2009
Message:
Corrected name of test case.
----
Added : /trunk/tests/test-kernel1d.args (Copy from path: /trunk/tests/test-kernel2d.args, Revision, 81)
Added : /trunk/tests/test-kernel1d.out (Copy from path: /trunk/tests/test-kernel2d.out, Revision, 81)
Added : /trunk/tests/test-kernel1d.syn (Copy from path: /trunk/tests/test-kernel2d.syn, Revision, 81)
Deleted : /trunk/tests/test-kernel2d.args
Deleted : /trunk/tests/test-kernel2d.out
Deleted : /trunk/tests/test-kernel2d.syn

Revision: 89
Author: jonwilliford
Date: 7:28:22 PM, Saturday, May 09, 2009
Message:
There was an error that made it so that only the last activation functions could be found, which is now fixed. Also fixed the same mistake with the kernel functions. Added functions that insure the that activation functions fail if not defined in order and fail otherwise.
----
Modified : /trunk/src/printer.ml
Modified : /trunk/src/translate1.ml
Added : /trunk/tests/fail-afun-order.args
Added : /trunk/tests/fail-afun-order.syn
Added : /trunk/tests/test-afun-chain.args
Added : /trunk/tests/test-afun-chain.out
Added : /trunk/tests/test-afun-chain.syn

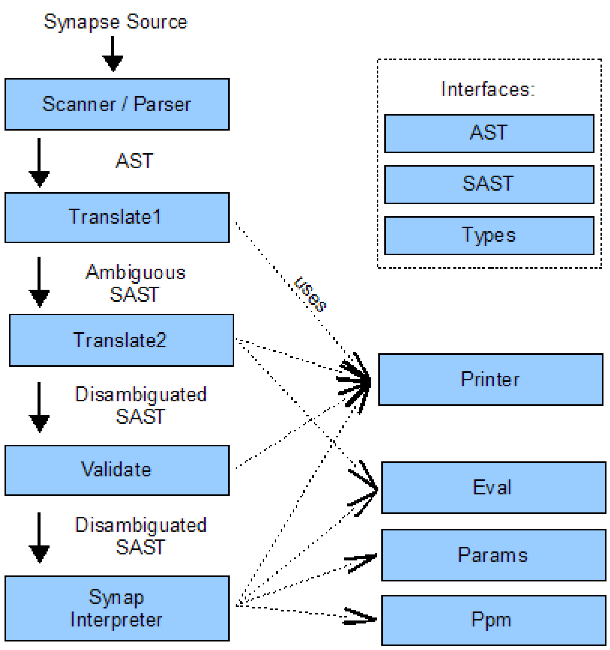
Revision: 90
Author: jonwilliford
Date: 6:40:56 PM, Monday, May 11, 2009
Message:
Minor fix to parser and just added comment to test.
----

```
Modified : /trunk/src/parser.mly
Modified : /trunk/tests/fail-afun-order.syn

Revision: 91
Author: jonwilliford
Date: 6:41:38 PM, Monday, May 11, 2009
Message:

----
Modified : /trunk/docs/Synapse Language Project Report.pdf
Modified : /trunk/docs/Synapse Language Project Report.tex
Modified : /trunk/docs/Synapse Language Refence Manual.pdf
Modified : /trunk/docs/Synapse Language Refence Manual.tex
Modified : /trunk/docs/Synapse_LRM_Chapter.tex
Modified : /trunk/docs/Synapse_Report_Language_Tutorial.tex
Modified : /trunk/docs/Synapse_Report_Lessons_Learned.tex
Modified : /trunk/docs/Synapse_Report_Project_Plan.tex
Modified : /trunk/docs/Synapse_Report_Test_Plan.tex
Modified : /trunk/docs/Synapse_Report_Tests.tex

Revision: 92
Author: jonwilliford
Date: 6:07:44 PM, Wednesday, May 13, 2009
Message:

----
Modified : /trunk/docs/Synapse Language Project Report.pdf
Modified : /trunk/docs/Synapse Language Project Report.tex
Modified : /trunk/docs/Synapse_LRM_Chapter.tex
Modified : /trunk/docs/Synapse_Report_Architectural_Design.tex
Modified : /trunk/docs/Synapse_Report_Language_Tutorial.tex
Modified : /trunk/docs/Synapse_Report_Tests.tex
Added : /trunk/docs/images
Added : /trunk/docs/images/barb-128x128.png
Added : /trunk/docs/images/barb-gamma.png
Modified : /trunk/src/eval.ml
Modified : /trunk/src/translate1.ml
Modified : /trunk/src/translate2.ml
Added : /trunk/tests/ex-gammacorrection.args
Added : /trunk/tests/ex-gammacorrection.syn
Added : /trunk/tests/fail-in-out-param.args
Added : /trunk/tests/fail-in-out-param.syn
Added : /trunk/tests/images/barb-128x128.ppm
Added : /trunk/tests/ones.txt

Revision: 93
Author: jonwilliford
Date: 8:57:26 PM, Wednesday, May 13, 2009
Message:

----
Modified : /trunk/docs/Synapse Language Project Report.pdf
Modified : /trunk/docs/Synapse Language Refence Manual.pdf
```

```
Modified : /trunk/docs/Synapse Language Refence Manual.tex
Modified : /trunk/docs/Synapse_LRM_Chapter.tex
Modified : /trunk/docs/Synapse_Report_Architectural_Design.tex
Modified : /trunk/docs/Synapse_Report_Language_Tutorial.tex
Modified : /trunk/docs/Synapse_Report_Lessons_Learned.tex
Modified : /trunk/docs/Synapse_Report_Tests.tex
Modified : /trunk/src/translate1.ml
Modified : /trunk/src/translate2.ml
Added : /trunk/tests/test-cos.args
Added : /trunk/tests/test-cos.out
Added : /trunk/tests/test-cos.syn
Added : /trunk/tests/test-exp.args
Added : /trunk/tests/test-exp.out
Added : /trunk/tests/test-exp.syn
Added : /trunk/tests/test-flip5c.args
Added : /trunk/tests/test-flip5c.out
Added : /trunk/tests/test-flip5c.syn
Added : /trunk/tests/test-sin.args
Added : /trunk/tests/test-sin.out
Added : /trunk/tests/test-sin.syn
```

# Chapter 5

# Architectural Design




I designed the project to be modular so that it would be easier to organize the code. Each box in the above diagram corresponds to an Objective Caml source file, with the exception of the scannar/parser which corresponds to two Object Caml source files. The scanner and parser performs the lexical analysis needed to convert the source files into the abstract syntax tree (AST). Translate1 performs the initial semantic analysis to create the semantically checked syntax tree (SAST), however it does not resolve the dimensionality of the neurons or expressions. Translate2 propagates the dimensionality from the parameters and neurons that have been explicitly given to the rest of the parameters and neurons. If it stops being able to disambiguate the program when there are still ambiguous expressions, then an error is thrown.

Once the SAST is disambiguated by Translate2, Validate runs some checks, mainly to check that there isn't anything that is still ambiguous. Initially I intended Validate to do more, including checking the result of Translate1. The validated SAST is then used by Synap to run the program with the help of Ppm (for reading in PPM images), Params (which includes code specific to handling parameters), and Eval (which performs the expression evaluations).

The SAST interface defines the expression tree as being a combination of expr_detail and a type defined in Types. The types allowed are:

1. Int - expressions in indices must have this type.
2. Matrix - floats and multidimensional have this type. This type specifies the dimensionality as a list of ints.

Floats are specified with a list of a single element of “1.”

3. MatrixUnknownSize - this is the type given to ambiguous expressions where the size is not known. This can only be in the SAST between the Translate1 and Translate2

# Chapter 6

# Test Plan

There are two main stages of my test plan. During the first stage, the development stage, I created a test before adding any functionality and committed the test along with the changes that added the functionality. I also added test cases that should fail when I recognized that cases that should fail. In the next stage I spent more time trying to correlate the tests with the rules. Footnotes have been added to the language reference manual that point to the corresponding test cases.

No code coverage was used. The test suite could be significantly improved if code coverage was assured. If it was used, it may also benefit the manual by making sure that every case or virtually every case was unambiguously specified in the language.

The test suite is in the Appendix B.

The automation was achieved by a script called testall.sh, which is a modified version of Dr. Stephen Edwards' script of the same name.

There are two types of tests, "success" tests and "failure" tests. The former makes up the majority of the tests. These tests should run successfully and the output should match with the tests predefined output (with extension ".out"). The failure tests should fail and are successful when the program throws an exception. Both test cases have an extension ".args" which define the file for the input and output parameters.

The entire test suite can be tested with the "make test" command in the src directory.

# Chapter 7

# Lessons Learned

The thing that struck me the most is how much better OCaml is for writing compilers than classical procedural programming languages. That being said, there seems to be many things I have find frustrating about OCaml. It can often be difficult to debug compile errors because partial function applications are allowed. For example, it took me about 4 hours to discover that a bug that I was getting was due "-1" being interpreted as two arguments, a function and an integer. I needed to use "(-1)" to fix the bug.

In order to utilize OCaml most efficiently and to reduce duplicate code, more thought needs to be put into the structure of data than other languages (although perhaps less than the case of complex inheritance situations). One simple example is using "Binop" to express all binary expressions so that a single function can often handle all cases. Also if two data structures share a common set of variables, such as activation and kernel functions, then delegating these variables to another type can a single function to be written to process this data rather than two.

I should have gotten the CheckFail() function in the testall.sh script to work sooner than what I did. As a result, most of my tests are just check for successful results. Having a test suite helped me on several occasions avoid messing up functionality that previously worked. This was helpful because often the early tests could be achieved with simplifying assumptions that would later require more logic.

Overall, it seemed easier to develop my own language than what I had previously thought. After getting the initial version up and running, new functionality could generally be added in a day. Even fairly significant refactoring was fairly easy.

# Appendix A

# Compiler Source

## A.1 scanner.mll

```
{ open Parser }

rule token = parse
  [' ' '\t' '\r' '\n'] { token lexbuf }
| "/*"      { comment lexbuf }            (* Comments *)
| '+' { PLUS }
| '-' { MINUS }
| '*' { TIMES }
| '/' { DIV }
| ';' { SEMI }
| ':' { COLON }
| ',' { COMMA }
| "<<" { LDIRECT }
| ">>" { RDIRECT }
| "=" { EQUALS }
| '^' { POW }
| '.' { DOT }
| '[' { LBRACK }
| ']' { RBRACK }
| '{' { LBRACE }
| '}' { RBRACE }
| '(' { LPAREN }
| ')' { RPAREN }
| "module" { MODULE }
| "size" { SIZE }
| "for" { FOR }
(*| "begin" { BEGIN } *)
| "end" { raise (Failure ("'end' is a reserved for future versions of Synapse."))
  }
| "t" { raise (Failure ("'t' is a reserved for future versions of Synapse.")) }
| "pi" { PI }
| "e" { E }
| "sqrt" { SQRT }
| "exp" { EXP }
| "sin" { SIN }
| "cos" { COS }
```

```
| "kernel" { KERNEL }
| "pragma" { PRAGMA }
| "input" { INPUT }
| "**" { CONV }
| ['a'-'z' 'A'-'Z'] ['a'-'z' 'A'-'Z' '0'-'9' '_']* as lit { ID( lit) }
| '$' ['0'-'9'] as lit { PARAM( int_of_char lit.[1]-48) }
| ['0'-'9']+ as lit { INT(int_of_string lit) }
| (['0'-'9']+ '.' ['0'-'9']* | ['0'-'9']* '.' ['0'-'9']+ )
    as lit { FLOAT(float_of_string lit) }
| eof { EOF }
| _ as char { raise (Failure("illegal character " ^ Char.escaped char)) }

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

## A.2 parser.mly

```
%{ open Ast
   let t1of3 (x,_,_) = x
   let t2of3 (_,x,_) = x
   let t3of3 (_,_,x) = x
%}

%token PLUS MINUS TIMES DIV SEMI COLON COMMA EOF
%token POW CONV
%token PI E SQRT EXP SIN COS
%token MODULE SIZE FOR T KERNEL PRAGMA INPUT /* END */
%token LDIRECT RDIRECT EQUALS
%token LPAREN RPAREN
%token LBRACE RBRACE LBRACK RBRACK DOT
%token <float> FLOAT
%token <int> INT
%token <string> ID
%token <int> PARAM

%left LDIRECT
%left PLUS MINUS
%left TIMES DIV
%left CONV
%left POW
%nonassoc UMINUS

%start program
%type < Ast.expr> expr
%type < Ast.expr> const_int_expr
%type < Ast.expr list > dimensions
%type < Ast.expr> lexpr
%type < Ast.program> program
%type < Ast.expr > for_expr
%type < string list > id_list
```

```

%%

program:
/* nothing */
{{ p_mdef=[]; p_mdecl=[]; p_adef=[]; p_wdef=[]; p_synap=[]; p_prag=[]; p_indim=[]
   }}
| program module_def
   {{ p_mdef = $2::$1.p_mdef; p_mdecl = $1.p_mdecl; p_adef = $1.p_adef; p_wdef =
         $1.p_wdef; p_synap = $1.p_synap; p_prag = $1.p_prag; p_indim =
                 $1.p_indim  }}
| program module_decl
   {{ p_mdef = $1.p_mdef; p_mdecl = $2::$1.p_mdecl; p_adef = $1.p_adef; p_wdef =
         $1.p_wdef; p_synap = $1.p_synap; p_prag = $1.p_prag; p_indim =
                 $1.p_indim }}
| program activation_def
   {{ p_mdef = $1.p_mdef; p_mdecl = $1.p_mdecl; p_adef = $2::$1.p_adef; p_wdef =
         $1.p_wdef; p_synap = $1.p_synap; p_prag = $1.p_prag; p_indim =
                 $1.p_indim }}
| program wght_def
   {{ p_mdef = $1.p_mdef; p_mdecl = $1.p_mdecl; p_adef = $1.p_adef; p_wdef =
         $2::$1.p_wdef; p_synap = $1.p_synap; p_prag = $1.p_prag; p_indim =
                 $1.p_indim }}
| program ext_synap_def
   {{ p_mdef = $1.p_mdef; p_mdecl = $1.p_mdecl; p_adef = $1.p_adef; p_wdef =
         $1.p_wdef; p_synap = $2::$1.p_synap; p_prag = $1.p_prag; p_indim =
                 $1.p_indim }}
| program pragma
   {{ p_mdef = $1.p_mdef; p_mdecl = $1.p_mdecl; p_adef = $1.p_adef; p_wdef =
         $1.p_wdef; p_synap = $1.p_synap; p_prag = $2::$1.p_prag; p_indim =
                 $1.p_indim }}
| program input_decl
   {{ p_mdef = $1.p_mdef; p_mdecl = $1.p_mdecl; p_adef = $1.p_adef; p_wdef =
         $1.p_wdef; p_synap = $1.p_synap; p_prag = $1.p_prag; p_indim =
                 $2::$1.p_indim }}

module_def:
  MODULE ID neurons RDIRECT neurons LBRACE module_body RBRACE
  {{ mod_name= $2; mod_inputs= $3; mod_outputs= $5;
  mod_neurons= t1of3 $7;
  mod_moddecls = t2of3 $7;
  mod_synaps = t3of3 $7 }}

pragma:
  PRAGMA LPAREN ID COMMA expr RPAREN SEMI { ($3,$5) }

input_decl:
  INPUT PARAM dimensions SEMI { ($2,$3) }

module_body:
        /* nothing */                  { ([],[],[]) }
| module_body neuron_def    { ( $2 :: t1of3 $1, t2of3 $1, t3of3 $1 ) }
| module_body module_decl   { ( t1of3 $1, $2 :: t2of3 $1, t3of3 $1 ) }
```

```
| module_body ext_synap_def      { ( t1of3 $1, t2of3 $1, $2 :: t3of3 $1 ) }


neuron_def:
 ID opt_dimensions LDIRECT synap_def SEMI
 {{ neuro_name=$1; neuro_ind=$2; neuro_syn=$4 }}

synap_def:
        expr {{ s_expr=$1; s_for=[] }}
      | expr FOR for_list {{ s_expr=$1; s_for=$3 }}

ext_synap_def:
 external_ref LDIRECT synap_def SEMI
        {{ s_dest=$1; s_syn=$3}}

external_ref:
  PARAM opt_dimensions         { Param($1,$2) }
| ID DOT ID opt_dimensions  { ExtNeuron( $1,[], $3,$4 ) }

for_list:
  for_expr              {[$1]}
| for_list for_expr     { $2 :: $1 }

for_expr:
  ID EQUALS LBRACK index_expr RBRACK    { ForExpr($1,$4)}

opt_dimensions:
  /* nothing */                 { [] }
| dimensions                    {($1)}

dimensions:      /* uses of dimensions from *declarations* cannot have ranges, BEGIN, or END */
  LBRACK index_list RBRACK      { List.rev $2 }

index_list:
  index_expr                    { [$1] }
| index_list COMMA index_expr      { $3::$1 }

index_expr:
  index_num                                              { ($1)}
  /*
| COLON                                                  {
        Span(Intgr(1),Intgr(1),End) } */
| index_num COLON index_num                              { Span($1,Intgr(1),$3) }
| index_num COLON const_int_expr COLON index_num        { Span($1,$3,$5) }

index_num:
  const_int_expr                        { ($1) }
/*| END                                    { End } */

const_int_expr:
  INT                                   { Intgr($1) }
| ID                                    { Id($1) }
```

```
| LPAREN const_int_expr RPAREN          { ($2) }
| const_int_expr PLUS   const_int_expr  { Binop($1, Add, $3) }
| const_int_expr MINUS  const_int_expr  { Binop($1, Sub, $3) }
| const_int_expr TIMES  const_int_expr  { Binop($1, Mul, $3) }
| MINUS const_int_expr %prec UMINUS     { Negate($2) }
| SIZE LPAREN lexpr COMMA INT RPAREN    { SizeOf($3,$5) }

module_decl:
 ID ID opt_dimensions SEMI   {{ modcl_type=$1; modcl_name=$2; modcl_dims=$3 }}

activation_def:
  ID LPAREN ID opt_fparams RPAREN EQUALS expr SEMI
        {{ act_name=$1; act_params=$4; act_local= $3; act_expr=$7 }}

wght_def:
  KERNEL ID LPAREN id_list opt_fparams RPAREN EQUALS expr SEMI
      {{ wght_name=$2; wght_params=$5; wght_ind= List.rev $4; wght_expr=$8 }}

opt_fparams:
  /* nothing */                { [] }
| SEMI fparams                       { ($2) }

fparams:
  fparam_list { List.rev $1 }

fparam_list:
  ID EQUALS FLOAT              { [ ($1,$3)] }
| ID EQUALS INT                { [ ($1,float_of_int $3)] }
| fparam_list COMMA ID EQUALS FLOAT     { ($3,$5) :: $1 }
| fparam_list COMMA ID EQUALS INT     { ($3,float_of_int $5) :: $1 }

neurons:
  neuron_list { List.rev $1 }

neuron_list:
  ID opt_dimensions { [($1,$2)] }
| neuron_list COMMA ID opt_dimensions { ($3,$4)::$1 }

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

expr:
  lexpr                        { ($1) }
| LPAREN expr RPAREN           { ($2) }
| expr PLUS   expr             { Binop($1, Add, $3) }
| expr MINUS  expr             { Binop($1, Sub, $3) }
| expr TIMES  expr             { Binop($1, Mul, $3) }
| expr DIV    expr             { Binop($1, Div, $3) }
| expr POW    expr             { Binop($1, Pow, $3) }
| MINUS expr %prec UMINUS      { Negate($2) }
| FLOAT                        { Float($1) }
| INT                          { Intgr($1) }
```

```
| ID LPAREN expr RPAREN         { ActRef($1,$3,[]) }
| ID LPAREN expr SEMI fparams  RPAREN       { ActRef($1,$3, $5) }
| expr CONV wghtref             { Conv($1,$3) }
| PI                            { Float(3.141592653589793) }
| E                             { Float(2.718281828459045) }
| EXP LPAREN expr RPAREN        { Exp($3) }
| SIN LPAREN expr RPAREN        { Sin($3) }
| COS LPAREN expr RPAREN        { Cos($3) }

lexpr:
  ID                     { Id($1) }
| ID dimensions          { NeuronRef($1,$2) }
| external_ref           { ExternalRef($1) }

wghtref:
  ID LPAREN RPAREN              { WghtRef($1,[]) }
| ID LPAREN fparams RPAREN  { WghtRef($1,$3) }
```

## A.3 ast.mli

```
type operator = Add | Sub | Mul | Div | Pow

type expr =
    Binop of expr * operator * expr
  | Float of float
  | Intgr of int
  | SizeOf of expr * int
  | Id of string
  | Negate of expr
  | NeuronRef of string * expr list
  | ExternalRef of external_ref
  | ActRef of string * expr * (string * float) list
  | WghtRef of string * (string * float) list
  | Conv of expr * expr
  | Exp of expr
  | Sin of expr
  | Cos of expr
(*  | End *)
  | Span of expr * expr * expr
  | ForExpr of string * expr
and external_ref =
  ExtNeuron of string * expr list * string * expr list
  | Param of int * expr list

type synap_def = {
  s_expr     : expr;
  s_for      : expr list; (* each expr must be ForExpr *)
}
type neuro_def = {
  neuro_name : string;
  neuro_ind : expr list;
```

```
  neuro_syn   : synap_def;
}
type mod_decl = {
  modcl_type : string;
  modcl_name : string;
  modcl_dims : expr list;
}
type ext_synap_def = {
  s_dest     : external_ref;
  s_syn      : synap_def;
}
type mod_def = {
  mod_name : string;
  mod_inputs : (string * expr list) list;
  mod_outputs : (string * expr list) list;
  mod_neurons : neuro_def list;
  mod_moddecls: mod_decl list;
  mod_synaps   : ext_synap_def list;
}
type act_def = {
  act_name : string;
  act_params : (string * float) list;
  act_local: string; (* the name of the input variable *)
  act_expr : expr;
}
type wght_def = {
  wght_name : string;
  wght_params : (string * float) list;
  wght_ind   : string list;
  wght_expr : expr;
}
type program = {
  p_mdef   : mod_def list;
  p_mdecl : mod_decl list;
  p_adef   : act_def list;
  p_wdef   : wght_def list;
  p_synap : ext_synap_def list;
  p_prag   : (string * expr) list;
  p_indim : (int * (expr list)) list;
}
```

## A.4 sast.ml

```
open Types
open Bigarray

let t1of3 (x,_,_) = x
let t2of3 (_,x,_) = x
let t3of3 (_,_,x) = x

module StringMap = Map.Make(String)
```

```

module IntMap = Map.Make(struct type t = int
        let compare x y = Pervasives.compare x y end)

let intMapCount m = IntMap.fold (fun k d a -> a+1) m 0


type operator = Add | Sub | Mul | Div | Pow
type operator1 = Exp | Sin | Cos | Negate

type mod_ref = string
type var_ref = string

type expr_detail =
    Binop of expr * operator * expr
  | Unop of operator1 * expr
(*  | Exp of expr
  | Sin of expr
  | Negate of expr
  | Cos of expr *)
  | Float of float
  | Intgr of int
  | SizeOf of expr_detail * int
  | Id of var_ref
  | ActRef of int * string * expr * (string * float) list
  | WghtRef of int * string * (string * float) list
  | Conv of expr * expr
  | Span of expr_detail * expr_detail * expr_detail
(*  | Begin
  | End *)
  | ForExpr of string * expr_detail
  | ExtNeuron of int * (* gsid *)
      mod_ref * expr_detail list * var_ref * expr_detail list
  | Param of int * expr_detail list
  | NeuronRef of int * (* gsid *)
      string * expr_detail list (* was Indx *)
and
  expr = expr_detail * Types.t

type ptype = InParam | OutParam

type channel =
    OpenIn of in_channel
 |  OpenOut of out_channel
 |  Closed
type param_io =
    SpaceDelimFile of string * channel
  | InPpm of string * int (* fname, time remaining *)
  | OutPpmSeq of string  (* fname pattern *)

type param_data =
    FloatArray of float array
  | ImageC3 of (int,Bigarray.int8_unsigned_elt,Bigarray.c_layout) Array3.t
```

```
  | Unallocated


type param_def = {
   pa_num: int;
   pa_type: Types.t;
   pa_ptype: ptype;
   pa_io: param_io;
   pa_data: param_data;
}
type actfun_def = {
   af_gsid: int; (* global symbol id *)
   af_name: string;
   af_invar: string;
   af_params: (string * float) list;
   af_expr: expr;
}
type kerfun_def = {
   kf_gsid: int; (* global symbol id *)
   kf_name: string;
   kf_indices: string list;
   kf_params: (string * float) list;
   kf_expr: expr;
}
type modtrix = {
   mt_dims: int list;
}
type neuron = {
  t: Types.t;
  v: float array array; (* 0 = working value, 1 = curr val, 2 = t-1 value *)
  n_name: string; (* only needed for debug *)
  n_mname: string; (* only needed for debug *)
}

type synapse = {
   sdest : expr_detail; (* ExtNeuron, Param, or NeuronRef *)
   sexpr : expr;
   sfor  : expr_detail list;
}

type prog = {
   syn: synapse IntMap.t; (* what is the key? pnum of param? *)
   pneurons: neuron IntMap.t; (* key is gsid *)
   pparams: param_def IntMap.t;
   actfuns: actfun_def IntMap.t;
   kerfuns: kerfun_def IntMap.t;
}
type neuro_type =
    InputNeuron | OutputNeuron | InterNeuron

type neuro_def = string * int * Types.t * neuro_type

type neuro_def_match =
```

```
    NeuroDefMatch of neuro_def
  | NoNeuroDefMatch

type mod_def = {
     m_name: string;
     m_inputs: neuro_def list;
     m_outputs: neuro_def list;
     m_neurons: neuro_def list; (* inter-neurons *)
     (*m_moddecls: mod_def list; *)
     m_synaps: synapse list; (* ExtNeuron or Neuron only - list just for debug purposes *)
}

let string_of_t = function
  Types.Int -> "Int"
| Types.Matrix( li ) ->
        "Matrix " ^ ( List.fold_left
        (fun str i -> match str with "" -> string_of_int i |
                                     str -> str ^ "x" ^ string_of_int i ) "" li)
| Types.MatrixUnknownSize -> "Matrix Unknown Size"

let print_t t = print_endline (string_of_t t)
```

## A.5 validate.ml

```
open Ast
open Sast
open Types
open Bigarray

(*
(* Make sure that:
 *
 *)
let rec Ast_expr = function


(* Make sure that:
 * The size of inputs are defined.
 *)
let rec Sast1_expr = function
 *)

exception ExceptionS2 of string;;


let str_of_ed_type = function
   Sast.Float(_) -> "Sast.Float"
 | Sast.Binop(_,_,_) -> "Sast.Binop"
 | Sast.Unop(op,_) ->
      ( match op with
```

```
          Sast.Sin -> "Sast.Sin"
        | Sast.Cos -> "Sast.Cos"
        | Sast.Exp -> "Sast.Exp"
        | Sast.Negate -> "Sast.Negate"
    )
 | Sast.Intgr(_) -> "Sast.Intgr"
 | Sast.SizeOf(_,_) ->"Sast.SizeOf"
 | Sast.Id(_) ->"Sast.Id"
 | Sast.NeuronRef(_,_,_) ->"Sast.NeuronRef"
 | Sast.ExtNeuron(_,_,_,_,_) ->"Sast.ExtNeuron"
 | Sast.Param(_,_) ->"Sast.Param"
 | Sast.ActRef(_,_,_,_) ->"Sast.ActRef"
 | Sast.WghtRef(_,_,_) ->"Sast.WghtRef"
 | Sast.Conv(_) ->"Sast.Conv"
 | Sast.Span(_,_,_) ->"Sast.Span"
 | Sast.ForExpr(_,_) ->"Sast.ForExpr"

let sast2str (e,t) = str_of_ed_type e

(* Need to make sure:
 * there are no macros (for, sizeof)
 * all dimensions of modules, neurons, and outputs are defined
 * no modules(?)
 *)
let rec sast2_expr = function
   exprd, Types.Matrix([]) ->
     print_endline (str_of_ed_type exprd);
     assert false
 | Sast.Unop(_,x1), t -> sast2_expr x1
 | Sast.Binop(x1,_,x2), t -> sast2_expr x1; sast2_expr x2
 | Sast.Conv(x1,x2), Types.MatrixUnknownSize ->
     raise( ExceptionS2 "Sast2.Binop has unresolved size.")
 | Sast.Conv((x1,t1),(x2,t2)), Types.Matrix(diml) ->
     assert( t1=t2 || ( t1=Types.Int && t2=Types.Matrix([1]) ) );
     sast2_expr (x1,t1);
     sast2_expr (x2,t2);
 | Sast.WghtRef(_,_,_),Types.Int -> assert false
 | Sast.WghtRef(_,_,_),Types.Matrix(diml) -> ()
 | Sast.WghtRef(_,_,_),Types.MatrixUnknownSize -> assert false
 | Sast.Conv(x1,x2), Types.Int ->  assert false
 | Sast.Intgr(_), Types.Int -> ()
 | Sast.Intgr(_), t -> raise( ExceptionS2 "Intgr has been associated with a non-int type.")
 | Sast.Float(_), Types.Matrix([1]) -> ()
 | Sast.Float(_), t -> raise( ExceptionS2 "Float has been associated with a non-float type (ie. single element matrix).")
 | Sast.SizeOf(exprd,_),Types.Int -> () (* more checks? *)
 | Sast.SizeOf(exprl,_),_ -> raise( ExceptionS2 "SizeOf has been associated with a non-int type.")
 | Sast.Id(vardecl), Types.MatrixUnknownSize -> raise( ExceptionS2 "Sast.Id has resolved size.")
 | Sast.Id(_),_ -> ()
 | Sast.Param(pnum,diml), Types.MatrixUnknownSize ->
```

```
      raise( ExceptionS2 ("Parameter "^(string_of_int pnum)^" has unresolved size."))
  | Sast.Param(pnum,diml), Types.Matrix(_) -> ()
  | Sast.Param(pnum,diml), _ -> raise( ExceptionS2 "Possible invalid Sast2 progrm")
  | Sast.ActRef(-1,_,_,_), _ -> raise( ExceptionS2 "An activation function reference is unresolved.")
  | Sast.ActRef(gsid,name,expr,fparamVals), Types.Matrix([1]) -> ()
  | Sast.ActRef(gsid,name,expr,fparamVals), Types.Matrix(_) ->
      raise( ExceptionS2 "Activation function must be scalar." )
  | Sast.ActRef(gsid,name,expr,fparamVals), _ -> raise( ExceptionS2 "Type/size of ActRef is unresolved.")
  | Sast.ExtNeuron(-1,mname,_,nname,_), _ ->
      raise( ExceptionS2
               (Printf.sprintf "Unresolve reference to external neuron %s.%s"
                 mname nname ))
  | Sast.ExtNeuron(_,_,_,_,_), Types.Matrix(_)-> ()
  | Sast.ExtNeuron(_,_,_,_,_), _ -> raise( ExceptionS2 "Type/size of ExtNeuron is unresolved.")
  | Sast.NeuronRef(-1,name,indl), _ ->
      raise( ExceptionS2
               (Printf.sprintf "Unresolve reference to local neuron '%s'." name ))
  | Sast.NeuronRef(gsid,name,indl), Types.Matrix(_) -> ()
  | exprd, Types.MatrixUnknownSize ->
       raise( ExceptionS2 ( (str_of_ed_type exprd)^" has unresolved size."))

 (*| _, Types.Matrix(_) -> () *)
  | n -> raise( ExceptionS2 ("Validation code not completed for "^(sast2str n)^"."))

let rec sast2_forexpr = function
    Sast.ForExpr(x1,x2) -> ()
  | e -> raise( ExceptionS2 ("Expected ForExpr but found "^(str_of_ed_type e)^"."))

let sast2_syn syn =
  sast2_expr syn.sexpr;
  List.iter sast2_forexpr syn.sfor

let sast2_param param =
  match param with
      { pa_type = Matrix(diml); pa_data = ImageC3(data) } ->
        let diml' = [ (Array3.dim1 data); (Array3.dim2 data); (Array3.dim3 data) ]
        in
          if diml' <> diml then
            raise( ExceptionS2 ("PPM parameter's array has an inconsistent size."))
    | { pa_type = Matrix(diml); pa_data = FloatArray(data) } -> ()
    | { pa_type = Matrix(diml); pa_data = Unallocated } ->
        raise( ExceptionS2 ("Parameter data unallocated."))
    | { pa_type = MatrixUnknownSize } -> raise( ExceptionS2 ("Parameter's size is unresolved."))
    | { pa_type = Int } -> raise( ExceptionS2 ("Parameter cannot be of type int."))
```

```


let sast2_neuron k n = match n with
    {t=Matrix(diml); v=v} ->
      let reqd_size = List.fold_left (fun a dim -> a * dim ) 1 diml in
        if Array.length v <> reqd_size then
          raise( ExceptionS2
                  (Printf.sprintf
                     "Neuron␣requires␣array␣of␣size␣%d,␣however␣has␣size␣of␣%d."
                     reqd_size (Array.length v)))
  | _ -> raise( ExceptionS2 ("Neuron␣must␣be␣of␣type␣Matrix"))

let sast2 p =
   IntMap.iter (fun k x -> sast2_syn x) p.syn;
   IntMap.iter (fun k x -> sast2_param x) p.pparams;
   IntMap.iter (fun k x -> sast2_neuron k x) p.pneurons;
   (* make sure every neuron is defined by a synapse *)
   IntMap.iter (fun gsid n ->
                  if IntMap.mem gsid p.syn = false then
                    raise( ExceptionS2
                            (Printf.sprintf
                               "Neuron␣%d␣isn't␣defined␣by␣any␣synapses!" gsid)))
     p.pneurons
```

## A.6 translate1.ml

```
open Ast
open Sast
open Types
open Params

exception Exception of string

type transl_env = {
    tparams: Sast.param_def IntMap.t;
    tactfuns: Sast.actfun_def IntMap.t;
    tkerfuns: Sast.kerfun_def IntMap.t;
    (*moddefs: mod_def StringMap.t; <- only needs to be used when module
    * declarations are allowed *)

    (* actualized module definitions *)
    tmods: Sast.mod_def StringMap.t;
}
let intMapCount m = IntMap.fold (fun k d a -> a+1) m 0
let stringMapCount m = StringMap.fold (fun k d a -> a+1) m 0

let neuron_count { m_inputs=n1; m_outputs=n2; m_neurons=n3 } =
  (List.length n1) + (List.length n2) + (List.length n3)

(* used to calculate the new "global symbol ids" or gsid *)
let symbol_count env =
```

```
  (intMapCount env.tparams) + (intMapCount env.tactfuns)
 + (StringMap.fold
      (fun k mdule count -> count + (neuron_count mdule))
      env.tmods
      0)

let create_environ (program:Ast.program) =
    let params = Params.create program in
       assert( IntMap.is_empty params = false );
    { tparams=params;
      tactfuns=IntMap.empty; tkerfuns=IntMap.empty;
      tmods=StringMap.empty; }

let ast2str = function
    Ast.Float(_) -> "Ast.Float"
  | Ast.Binop(_,_,_) -> "Ast.Binop"
  | Ast.Intgr(_) -> "Ast.Intgr"
  | Ast.SizeOf(_,_)->"Ast.SizeOf"
  | Ast.Id(_)->"Ast.Id"
  | Ast.Negate(_)->"Ast.Negate"
  | Ast.NeuronRef(_,_)->"Ast.NeuronRef"
  | Ast.ExternalRef(_)->"Ast.ExternalRef"
  | Ast.ActRef(_,_,_)->"Ast.ActRef"
  | Ast.WghtRef(_,_)->"Ast.WghtRef"
  | Ast.Conv(_)->"Ast.Conv"
  | Ast.Exp(_)->"Ast.Exp"
  | Ast.Sin(_)->"Ast.Sin"
  | Ast.Cos(_)->"Ast.Cos"
  | Ast.Span(_,_,_)->"Ast.Span"
  | Ast.ForExpr(_,_)->"Ast.ForExpr"

let rec translate expr env =
match expr with
     Ast.Float(v) -> Sast.Float(v), Types.Matrix([1])
   | Ast.Intgr(v) -> Sast.Intgr(v), Types.Int
   | Ast.Id(name) -> Sast.Id(name), Types.MatrixUnknownSize (* local function variable  or neuronref*)
   | Ast.Exp(e1) ->
       let e1' = translate e1 env in
         Sast.Unop( Sast.Exp, e1' ), Types.MatrixUnknownSize
   | Ast.Sin(e1) ->
       let e1' = translate e1 env in
         Sast.Unop( Sast.Sin, e1' ), Types.MatrixUnknownSize
   | Ast.Cos(e1) ->
       let e1' = translate e1 env in
         Sast.Unop( Sast.Cos, e1' ), Types.MatrixUnknownSize
   | Ast.Negate(e1) ->
       let e1' = translate e1 env in
         Sast.Unop( Sast.Negate, e1' ), Types.MatrixUnknownSize
   | Ast.Binop(e1,op,e2) ->
       let e1' = translate e1 env in
       let e2' = translate e2 env in
       let op' = match op with
```

```
          Ast.Add -> Sast.Add | Ast.Sub -> Sast.Sub | Ast.Mul -> Sast.Mul
        | Ast.Div -> Sast.Div | Ast.Pow -> Sast.Pow
    in
      Sast.Binop(e1',op',e2'), Types.MatrixUnknownSize
  | Ast.Conv(e1,e2) -> (* Should allow multiple Conv, but type doesn't *)
    let e1' = translate e1 env in
    let e2' = translate e2 env in (* e2 should be WghtRef *)
      Sast.Conv(e1',e2'), Types.MatrixUnknownSize
  | Ast.ExternalRef(extref) ->
    let _,expr,t = translate_extref extref env OutputNeuron in
      expr,t
  | Ast.WghtRef( name, fparams ) ->
    let gsid = IntMap.fold
                  (fun k {kf_name=kf_name} a ->
                     Printf.printf "trying to match %s %s\n" kf_name name;
                     if kf_name=name then
                       k
                     else a)
                  env.tkerfuns
                  (-1)
    in
      if gsid = -1 then
        raise
          (Exception
             (Printf.sprintf "Kernel function %s is undefined." name));
      Sast.WghtRef(gsid, name, fparams), Types.MatrixUnknownSize

  | Ast.ActRef(name,expr,fparams) ->
    let expr' = translate expr env in
    let gsid = IntMap.fold
                  (fun k {af_name=af_name} a ->
                     if af_name=name then
                       k
                     else a
                  )
                  env.tactfuns
                  (-1)
    in
      if gsid = -1 then
        raise
          ( Exception (Printf.sprintf "Activation function %s is undefined." name));
      Sast.ActRef(gsid, name, expr', fparams), Types.MatrixUnknownSize

  | Ast.ForExpr(_,_)->
    raise ( Exception ("Error: for expression in invalid environment. "))

  | Ast.NeuronRef(name,indl)->
    let (indl':Sast.expr_detail list) = List.fold_left
                    (fun li i -> (translate_ind i env) :: li)
                    [] indl
    in
      Sast.NeuronRef(-1,name,indl'), Types.MatrixUnknownSize
```

```
  | Ast.Span(_,_,_) ->
      raise ( Exception "Span is not allowed here!")
  | Ast.SizeOf(_,_) ->
      raise ( Exception "SizeOf is not allowed here!")

and translate_ind exprd env =  (* Inside index *)
  match exprd with
      Ast.Intgr(v) -> Sast.Intgr(v)
    | Ast.Id(name) -> Sast.Id(name) (* must refer to for-loop var *)
    | Ast.Negate(e1) ->
        let e1' = translate_ind e1 env in
          Sast.Unop(Sast.Negate, (e1',Types.Int))
    | Ast.Binop(e1,op,e2) ->
      let e1' = translate_ind e1 env in
      let e2' = translate_ind e2 env in
      let op' = match op with
           Ast.Add -> Sast.Add | Ast.Sub -> Sast.Sub | Ast.Mul -> Sast.Mul
         | Ast.Div -> raise( Exception "Error division is not allowed in indices.")
         | Ast.Pow -> raise( Exception "Power operators are not allowed in indices.")
      in
         Sast.Binop( (e1',Types.Int),op',(e2',Types.Int))
    | Ast.SizeOf(exprl,dim) ->
        let e',t = translate exprl env in
          Sast.SizeOf(e',dim)
    | Ast.Span(e1,step,e2) ->
        let e1' = translate_ind e1 env in
        let step' = translate_ind step env in
        let e2' = translate_ind e2 env
        in
          Sast.Span(e1',step',e2')
      (* the following errors should have been caught as syntax errors *)
    | Ast.ForExpr(_,_) -> assert false
    | Ast.Cos(_) -> assert false
    | Ast.Sin(_) -> assert false
    | Ast.Exp(_) -> assert false
    | Ast.Conv(_,_) -> assert false
    | Ast.WghtRef(_,_) -> assert false
    | Ast.ActRef(_,_,_) -> assert false
    | Ast.ExternalRef(_) -> assert false
    | Ast.NeuronRef(_,_) -> assert false
    | Ast.Float(_) -> assert false

and translate_extref eref env reqd_nt = match eref with
    Ast.Param(pnum,diml) ->
      let diml' = List.rev
                    (List.fold_left
                       (fun li dim -> (translate_ind dim env)::li )
                       [] diml)
      in
          pnum,Sast.Param(pnum,diml'), Types.MatrixUnknownSize
  | Ast.ExtNeuron(mname,mindl,nname,nindl) ->
      Printf.printf "Ast.ExtNeuron(%s,...,%s,...)\n" mname nname;
```

```
       if StringMap.mem mname env.tmods = false then
         raise (Exception
                  ("Could not find "^mname^" module."));
       let gsid =
           let neurons =
             let tmod = StringMap.find mname env.tmods in
               List.append tmod.m_inputs (List.append tmod.m_outputs tmod.m_neurons)
           in
             List.fold_left
               (fun a (name,gsid,_,nt) ->
                  if name = nname then
                    (
                      if nt<>reqd_nt then match nt with
                          InterNeuron -> raise (Exception
                             (Printf.sprintf
                               "Attempted to reference interneuron '%s' outside of module."
                               name
                             ))
                        | InputNeuron -> raise (Exception
                             (Printf.sprintf
                               "Attempted to receive action potentials from input neuron '%s' outside of module."
                               name
                             ))
                        | OutputNeuron -> raise (Exception
                             (Printf.sprintf
                               "Attempted to send action potentials to output neuron '%s' outside of module."
                               name
                             ))
                      else gsid
                    )
                  else
                    a
               )
               (-1)
               neurons
       in
         if gsid= -1 then raise (Exception (Printf.sprintf "Could not find the neuron %s in %s." nname mname));
       let translate' indl =
         List.fold_left
           (fun indl indx -> (translate_ind indx env) :: indl )
           [] indl
       in
         gsid,
        Sast.ExtNeuron( gsid, mname,
                       (translate' mindl),
                       nname,
                       (translate' nindl)),
        Types.MatrixUnknownSize
```

```

let translate_for expr env = match expr with
    Ast.ForExpr(varname,e1)->
      let e1' = translate_ind e1 env
      in
        Sast.ForExpr(varname,e1'), env
  | Ast.ForExpr(_,e) ->
      raise ( Exception ("Error: invalid format for for-macro " ^
                         (ast2str e)^"."))
  | e -> raise( Exception ("Error: expected for-expression, found "^ (ast2str e)^"."))

let translate_syn synap env =
 let expr' = translate synap.s_expr env in
 let (sfor':Sast.expr_detail list),env' = List.fold_left
                      (fun (li,env') sfor ->
                         let ((fexpr:Sast.expr_detail),env') =
                           translate_for sfor env in fexpr::li,env' )
                      ([], env)
                      (synap.s_for: Ast.expr list)
 in
   (expr', sfor')

(* Performs semantic analysis AST->SAST for synapses, some sizes may still be
 * undetermined *)
let translate_s synap env =
 let (gsid,extref',_) = translate_extref synap.s_dest env InputNeuron in
 let (expr',sfor') = translate_syn synap.s_syn env
 in
   gsid,{ sdest=extref'; sexpr=expr'; sfor=sfor'},env


(* Dumbed-down version of Eval.evali that cannot use SizeOf or Id *)
let rec evalc = function
    Sast.Intgr( v ) -> v
  | Sast.Binop( (e1,_), op, (e2,_) ) ->
      (
      let v1 = evalc e1 and v2 = evalc e2 in
          match op with
            Sast.Add -> v1 + v2
          | Sast.Sub -> v1 - v2
          | Sast.Mul -> v1 * v2
          | Sast.Div -> raise( Exception "Division is not allowed in indices.")
          | Sast.Pow -> raise( Exception "Power operator is not allowed in indices.")
      )
  | e -> raise(Exception ( Printer.string_of_exprd e ^ " can not be used inside the dimension definition of a neuron or module."))

let match_local_neuron idname allneurons =
  List.fold_left
    (fun a (nname,gsid,t,nt) ->
       if nname=idname then NeuroDefMatch( (nname,gsid,t,nt) )
```

```
        else a)
      NoNeuroDefMatch
      allneurons

let rec convert_ids_to_neurorefs (exprd,t) forl allneurons =
  (convert_ids_to_neurorefs_d exprd forl allneurons),t
and convert_ids_to_neurorefs_d exprd forl allneurons = match exprd with
      Sast.Id(idname) ->
        Printf.printf "Trying to match '%s' with a neuron..." idname;
        let mneuron = match_local_neuron idname allneurons
        in
          (
          match mneuron with

            (* TO-DO: should make sure that there are no conflicts between neuron
            * reference and a for-var  *)
            Sast.NeuroDefMatch( (nname,gsid,t,nt) )->
              Printf.printf "success matching to %d.\n" gsid;
                  Sast.NeuronRef(gsid,nname,[])
          | NoNeuroDefMatch ->
              Printf.printf "failed.\n";
              Sast.Id(idname)
          )
    | Sast.NeuronRef( -1, nname, indl ) ->
        let mneuron = match_local_neuron nname allneurons
        in
          (
            match mneuron with
                Sast.NeuroDefMatch( (nname,gsid,t,nt) )->
                  Sast.NeuronRef(gsid,nname,indl)
              | NoNeuroDefMatch ->
                  raise (Exception (Printf.sprintf "Unable to resolve local neuron reference '%s'." nname))
          )

    | Sast.NeuronRef( a, b, c ) -> Sast.NeuronRef( a, b, c )
    | Sast.Binop(x1,op,x2) ->
        let x1' = convert_ids_to_neurorefs x1 forl allneurons in
        let x2' = convert_ids_to_neurorefs x2 forl allneurons in
          Sast.Binop(x1',op,x2')
    | Sast.Unop(op,x) ->
        Sast.Unop( op, convert_ids_to_neurorefs x forl allneurons )
    | Sast.Float(v) -> Sast.Float(v)
    | Sast.Intgr(v) -> Sast.Intgr(v)
    | Sast.SizeOf(exprd, d) -> Sast.SizeOf( (convert_ids_to_neurorefs_d exprd forl
      allneurons), d )
    | Sast.ActRef(a,b,x,c) ->
        let x' = convert_ids_to_neurorefs x forl allneurons in
          Sast.ActRef(a,b,x',c)
    | Sast.WghtRef( gsid, n, fparams ) ->
        Sast.WghtRef( gsid, n, fparams )
    | Sast.Conv( x1, x2 ) ->
        let x1' = convert_ids_to_neurorefs x1 forl allneurons in
```

```
      let x2' = convert_ids_to_neurorefs x2 forl allneurons in
        Sast.Conv( x1', x2' )
  | Sast.Span( a, b, c) -> Sast.Span( a, b, c ) (* neurorefs can't be in spans - except through sizeof *)
  | Sast.ForExpr( name, exprd ) -> Sast.ForExpr( name, exprd)(* except through sizeof *)
  | Sast.ExtNeuron( a, b, c, d, e ) ->
      Sast.ExtNeuron( a, b, c, d, e )
  | Sast.Param( pnum, indl ) -> Sast.Param( pnum, indl)



let translate_mdef mdef env synmap =
  let synmap,mdef' =
    let symc = symbol_count env in
    let conv_neurodefs symc pneurons nt =  (* converts Ast neurons to Sast neurons *)
      List.fold_left
        (fun (inplist,symc) (name,diml) ->
           let diml' =
             List.fold_left
               (fun diml' expr ->
                  (evalc (translate_ind expr env)) :: diml')
               []
               diml
           in
             (name,symc+1,Types.Matrix(diml'),nt) :: inplist,(symc+1))
        ([],symc)
        pneurons
    in
    let inputs',symc = conv_neurodefs symc mdef.mod_inputs InputNeuron in
    let outputs',symc = conv_neurodefs symc mdef.mod_outputs OutputNeuron in
    let (synaps',env) = List.fold_left  (* external references *)
                        (fun (synmap,env) syn ->
                           let (gsid,syn,env) = translate_s syn env
                           in (IntMap.add gsid syn synmap), env )
                        (synmap,env)
                        mdef.mod_synaps
    in
    let (neurons',synaps'',env,symc)
          = List.fold_left   (* local references *)
               (fun (neurli,synmap,env,symc) n ->
                  let gsid,nt = List.fold_left
                                  (fun a ((name,nid,_,nt):neuro_def) ->
                                     if name = n.neuro_name then
                                       nid,nt
                                     else a
                                  )
                                  (-1,InputNeuron)
                                  outputs'
                  in
                  let (symc',gsid) = if gsid = -1 then (symc+1,symc+1) else (symc,gsid) in
```

```
                  let (sexpr,sfor) = translate_syn n.neuro_syn env in
                  let nindl =
                    List.fold_left
                      (fun li ind -> (translate_ind ind env) :: li)
                      [] n.neuro_ind
                  in
                  let (n':Sast.neuro_def) = n.neuro_name,gsid,Types.MatrixUnknownSize,nt
                  in
                    n'::neurli,
                    (IntMap.add gsid { sdest=Sast.NeuronRef(gsid,n.neuro_name,nindl);
                     sexpr=sexpr; sfor=sfor} synmap),env,symc')
               ([],synaps',env,symc)
               mdef.mod_neurons
    in
    let synaps''' =
      let allneurons = List.append inputs' (List.append outputs' neurons')
        in
        IntMap.map (fun {sdest=sdest;sexpr=sexpr;sfor=sfor} ->
                        let sexpr' = (convert_ids_to_neurorefs sexpr sfor allneurons)
                        in
                          print_endline (Printer.string_of_expr sexpr');
                          { sdest=sdest; sexpr=sexpr'; sfor=sfor }) synaps''
    in
      IntMap.iter (fun k d -> print_endline (Printer.string_of_synap k d) ) synaps''';
      synaps''',
      { m_name=mdef.mod_name; m_inputs=inputs'; m_outputs=outputs'; m_neurons=neurons';
        m_synaps=IntMap.fold (fun k d li -> d :: li ) synaps''' [] }
  in let tmods' =
    StringMap.add mdef.mod_name mdef' env.tmods
  in
    synmap,{ tmods=tmods'; tparams=env.tparams;
             tactfuns=env.tactfuns; tkerfuns=env.tkerfuns }

(* translates actdef from Ast.program to transl_env  *)
let translate_adef actdef env = match actdef with
    { act_name = name; act_params=fparams;
      act_local = invar; act_expr=expr }
    ->
      Printf.printf "Translating activation function %s.\n" name;
      let expr' = translate expr env in
      let gsid = (intMapCount env.tparams) + (intMapCount env.tactfuns) + 1 in
      let adef' = { af_gsid = gsid; af_name=name; af_invar=invar;
          af_params=fparams; af_expr=expr' }
      in
      let adefs' = IntMap.add gsid adef' env.tactfuns in
        Printf.printf "Current activation functions:\n";
        IntMap.iter (fun gsid adef ->
                         print_endline (Printer.string_of_afundef gsid adef))
```

```
          adefs';
        { tparams=env.tparams; tactfuns=adefs';
          tkerfuns=env.tkerfuns;
          tmods=env.tmods }

(* translates kerneldef from Ast.program to transl_env  *)
let translate_kdef kerdef env = match kerdef with
    { wght_name = name; wght_ind = indices;
      wght_params = fparams; wght_expr = expr }
   ->
      let expr' = translate expr env in
      let gsid = (intMapCount env.tparams) +
                 (intMapCount env.tactfuns) +
                 (intMapCount env.tkerfuns) + 1
      in
      let kdef' = { kf_gsid=gsid; kf_name=name; kf_indices=indices;
                    kf_params=fparams; kf_expr=expr' }
      in
      let kdefs' = IntMap.add gsid kdef' env.tkerfuns
      in
        Printf.printf "kern %s\n" name;

        { tparams=env.tparams; tactfuns=env.tactfuns;
          tmods=env.tmods;
          tkerfuns=kdefs' }


let translate_p (program:Ast.program) env=
  let env = List.fold_left
              (fun env adef ->
                 let env' = translate_adef adef env in env')
              env
              (List.rev program.p_adef)
  in
  let env = List.fold_left
              (fun env kdef ->
                 let env' = translate_kdef kdef env in env')
              env
              program.p_wdef
  in
    Printf.printf "There are %d kernel definitions.\n"
      (intMapCount env.tkerfuns);
  let msynaps,env = List.fold_left
              (fun (synm,env) mdef -> translate_mdef mdef env synm )
              (IntMap.empty,env)
              program.p_mdef
  in
  let synapsel,env = List.fold_left
                     (fun (m,env) syn ->
                        let (gsid,syn,env) = translate_s syn env
                        in
                         if IntMap.mem gsid m then
```

```
                              raise( Exception (Printf.sprintf "A synapse is redefining the symbol with ID %d." gsid ));
                             (IntMap.add gsid syn m), env )
                     (msynaps,env)
                     program.p_synap
  in
  let neurons =
    StringMap.fold
      (fun mname mdef a -> match mdef with
           { m_inputs=inputs; m_outputs=outputs; m_neurons=inters } ->
             let addneurons (nlist:Sast.neuro_def list) nmap =
               List.fold_left
                 (fun nmap (name,gsid,t,nt) ->
                    Printf.printf "Creating neuron %s.%s with gsid %d\n" mname name gsid;
                    IntMap.add
                      gsid
                      { n_mname=mname; n_name=name;
                        t=t; v=[| [| 0.; 0. |] |] }
                      nmap
                 )
                 nmap
                 nlist
             in
               (addneurons inputs (addneurons outputs (addneurons inters IntMap.empty))))
      env.tmods
      IntMap.empty
   in { pparams=env.tparams; actfuns=env.tactfuns; kerfuns=env.tkerfuns;
        syn=synapsel; pneurons=neurons }

let program p =
  let env = create_environ p in
      assert( (IntMap.is_empty env.tparams) = false );
  let p = translate_p p env
  in
      IntMap.iter
        (fun k pam -> print_endline (Printer.string_of_param pam.pa_num pam))
        p.pparams;
      p
```

## A.7 translate2.ml

```
open Types
open Sast
open Printer
open Printf
open Bigarray
open Eval

exception Exception of string
```

```
exception UnresolvedParam

type resolve_env = {
    unres_symbols: int list;
}

let create_env (p:Sast.prog) =
      let unres = IntMap.fold
          (fun key param li ->
             match param with
               { pa_type=MatrixUnknownSize; pa_num=pa_num} -> pa_num :: li
             | _ ->  li )
          p.pparams []
      in
      let unres = IntMap.fold
          (fun gsid n li ->
             match n with
               { t=MatrixUnknownSize; } -> gsid :: li
             | _ ->  li )
          p.pneurons unres
      in
      { unres_symbols=unres }

(* Returns the number of symbols for which the sizes are unknown *)
let ambiguity { unres_symbols=unres_symbols } { syn=syn } =
    List.length unres_symbols +
    (IntMap.fold
       (fun key syn num -> match syn.sexpr with _,MatrixUnknownSize->num+1 | _->
          num ) syn 0)




(* Inside index *)
let rec retrieve_forvar = function
    Id(varn) -> [varn]
  | Intgr(_) -> []
  | SizeOf(_,_) -> [] (* for-variables are not allowed in sizeof *)
  | Binop((e1,t1),op,(e2,t2)) ->
      let varn1 = retrieve_forvar e1 in
      let varn2 = retrieve_forvar e2 in
        List.append varn1 varn2
  | Unop(op,(e1,t1)) ->
      (retrieve_forvar e1)
  | Span(e1,e2,e3) -> [] (* for-variables not allowed in Span *)
  | NeuronRef(_,_,_) -> assert false
  | Param(_,_) -> assert false
  | ExtNeuron(_,_,_,_,_) -> assert false
  | ForExpr(_,_) -> assert false
  | Conv(_,_) -> assert false
  | WghtRef(_,_,_) -> assert false
  | ActRef(_,_,_,_) -> assert false
  | Float(_) -> assert false
```

```

let rec resolve_looped_dim dexpr sfor =
  let varnl = retrieve_forvar dexpr in
  let varn = match varnl with
       [] -> raise( Exception
                        ("Index expression didn't contain for-loop"^
                         " - this may be supported in the future."))
     | [varn] -> varn
     | varn :: li ->
         raise( Exception
                   "Only a single for-variable can be in a single dimension!!!")
  in
   match sfor with
        [] -> raise( UnresolvedParam )
      | ForExpr(id,span) :: forlist ->
          ( if varn <> id then
              resolve_looped_dim dexpr forlist
            else
            (
              let (nums : int list) = Eval.nums_of_span span in
              let (i:int) = List.fold_left
                              (fun (x:int) (y:int) -> if (x > y) then x else y)
                                 (-1) nums
              in i
            )
          )
      | sast :: _ -> raise( Exception ("Error: Invalid expression type "^(
         string_of_exprd sast)^" in for-list!"))

let shorten_diml diml =
  match
      (fst
         (List.fold_right
            (fun d (li,do_reduce) ->
               if do_reduce && d = 1 then
                 (li,true)
               else
                 (d :: li,false))
            diml
            ([],true)
         )
      )
  with [] -> [1]
     | diml -> diml

type opt_int = Int of int | NoInt

let resolve_syn_dest sfor indl expr_diml =
  let diml' = List.map (fun d -> resolve_looped_dim d sfor) indl in
    List.iter (fun dim -> Printf.printf "%d," dim) diml'; print_endline "";
    (shorten_diml (List.append diml' expr_diml)) (* ???? *)

(* returns modified program and list of unresolved parameters *)
```

```
let resolve_symbol_ref p pnum =
  let synap = IntMap.find pnum p.syn in
    match synap with
        { sexpr=(sexprd,Types.MatrixUnknownSize); sfor=sfor} ->
          p,Int(pnum)      (* can't do anything, but push pnum as being unresolved *)
      | { sexpr=sexprd,Types.Int; sfor=sfor} ->
          assert false (* compiler error *)

      | { sdest=Param(pnum,indl); sexpr=sexprd,Types.Matrix(expr_diml); sfor=sfor } ->
          (
          print_endline (Printer.string_of_synap pnum synap);
          assert ( IntMap.mem pnum p.pparams = true );
          let pam = IntMap.find pnum p.pparams in
            match pam with
                { pa_type=Matrix(li) } -> p,NoInt (* nothing to do *)
              | { pa_type=MatrixUnknownSize } ->
                  (
                  try
                  let diml' = resolve_syn_dest sfor indl expr_diml in
                  let params = IntMap.add pnum
                                 { pa_num=pnum; pa_type=Types.Matrix( diml' );
                                   pa_ptype=pam.pa_ptype; pa_io=pam.pa_io; pa_data=Unallocated }
                                 p.pparams
                  in
                    { syn=p.syn; pparams=params;
                      actfuns=p.actfuns; pneurons=p.pneurons; kerfuns=p.kerfuns;
                    }, NoInt
                  with UnresolvedParam -> p, Int(pnum)
                  )
              | _ ->
                    raise ( Exception "For-loop not yet completed.")
          )

      | { sdest=NeuronRef(gsid,name,indl); sexpr=sexprd,Types.Matrix(expr_diml); sfor=sfor} ->
          (
          print_endline (Printer.string_of_synap pnum synap);
          assert ( IntMap.mem pnum p.pneurons = true );
          let n = IntMap.find pnum p.pneurons in
            match n with
                { t=Matrix(li) } -> p,NoInt (* nothing to do *)
              | { t=MatrixUnknownSize } ->
                  (
                  try
                  let diml' = resolve_syn_dest sfor indl expr_diml in
                  let neurons' = IntMap.add gsid
                                 { t=Types.Matrix( diml' );
                                   v=n.v;
                                   n_name = n.n_name; n_mname=n.n_mname }
                                 p.pneurons
```

```
                    in
                      { syn=p.syn; pparams=p.pparams;
                        actfuns=p.actfuns; kerfuns=p.kerfuns; pneurons=neurons' }, NoInt
                    with UnresolvedParam -> p, Int(pnum)
                    )
                | _ ->
                      raise ( Exception "For-loop not yet completed.")
            )
        | _ -> raise ( Exception "Parameter misassociated with external neuron reference.")

(* calculate the "span" of the dimension given the restriction of the index *)
let index_span exprd = match exprd with
    Sast.Span(e1,e2,e3) -> List.length (Eval.nums_of_span exprd)
  | _ -> 1

let rec eval_expr_diml indl diml p =
  match (indl, diml) with
      [],[] -> []
    | (ind :: indl'),(dim :: diml') ->
        (index_span ind) :: (eval_expr_diml indl' diml' p)
    | [],diml -> (shorten_diml diml)
    | _ ->
        raise ( Exception "Index specifies more dimensions than what is declared.")

(* returns the hopefully less ambiguious expression and the number
* of expressions that have had their types resolves *)
let rec resolve_expr expr p = match (expr:Sast.expr) with
    _,Types.Int -> expr,0
  | _,Types.Matrix(_) -> expr,0
  | Sast.Unop(op,e1), Types.MatrixUnknownSize ->
      let (e1',t1),amb1 = resolve_expr e1 p in
      let t',a = match t1 with
          Types.Int -> Types.Matrix([1]), 1
        | Types.Matrix(diml) -> Types.Matrix(diml),1
        | Types.MatrixUnknownSize -> Types.MatrixUnknownSize,0
      in
        (Sast.Unop(op,(e1',t1)), t'), (amb1+a)

  | Sast.Binop(e1,op,e2), Types.MatrixUnknownSize ->
      let (e1',t1),amb1 = resolve_expr e1 p in
      let (e2',t2),amb2 = resolve_expr e2 p in
      let t',a = match (t1,t2) with
          (Types.MatrixUnknownSize,_) -> Types.MatrixUnknownSize, 0 (* 0=amb resolution at current node *)
        | (_,Types.MatrixUnknownSize) -> Types.MatrixUnknownSize, 0
        | (Types.Matrix(diml),Types.Int) -> Types.Matrix(diml), 0
        | (Types.Int, Types.Matrix(diml)) -> Types.Matrix(diml), 0
        | (Types.Matrix(diml),Types.Matrix([1])) -> Types.Matrix(diml), 1
        | (Types.Matrix([1]),Types.Matrix(diml)) -> Types.Matrix(diml), 1
        | (t1,t2) ->
```

```
          if t1=t2 then t1,1
          else
            raise (Exception
                    (Printf.sprintf
                       "Binop has non-compatible types %s and %s."
                       (Printer.string_of_type t1)
                       (Printer.string_of_type t2)) )
    in
      print_endline (Printer.string_of_type t1);
      print_endline (Printer.string_of_type t2);
      (Sast.Binop((e1',t1),op,(e2',t2)), t'), (amb1+amb2+a)
  | Sast.Conv(e1,e2), Types.MatrixUnknownSize ->
    let (e1',t1),amb1 = resolve_expr e1 p in
    let (e2',t2),amb2 = resolve_expr e2 p in
    let t,t2',a = match (t1,t2) with
        (Types.MatrixUnknownSize,t) -> Types.MatrixUnknownSize, t, 0
      | (Types.Matrix(diml1),Types.MatrixUnknownSize) -> Types.Matrix([1]),Types.Matrix(diml1),2
      | (Types.Matrix(diml1),Types.Matrix(diml2)) ->
          assert (diml1=diml2);
          Types.Matrix([1]),Types.Matrix(diml1),1
      | (Types.Int, MatrixUnknownSize) -> Types.Matrix([1]),Types.Matrix([1]),2
      | (Types.Int, Matrix([1])) -> Types.Matrix([1]),Types.Matrix([1]),1
      | (Types.Int, _ ) -> assert false
      | ( _ ,Types.Int) -> assert false
    in
      (Sast.Conv((e1',t1),(e2',t2')), t), (amb1+amb2+a)

  | Sast.WghtRef(gsid,name,fparams), Types.MatrixUnknownSize -> (* resolution occurs by Conv not here*)
    (Sast.WghtRef(gsid,name,fparams),Types.MatrixUnknownSize), 0
  | Sast.ActRef(gsid,name,e1,fparams), Types.MatrixUnknownSize ->
    let (e1',t1),amb1 = resolve_expr e1 p in
    let t,a =
      (
      match t1 with
          Types.MatrixUnknownSize -> t1,0
        | Types.Matrix([1]) -> t1,1
        | Types.Matrix(diml) ->
            raise ( Exception "A non-scalar expression cannot be passed to an activation function.")
        | Types.Int -> assert false
      )
    in
      (Sast.ActRef(gsid,name,(e1',t1),fparams),t),(amb1+a)

  | Sast.Param(pnum,indl), Types.MatrixUnknownSize ->
    let pam = IntMap.find pnum p.pparams in
      (Printf.printf
         "Trying to resolve ambiguity of $%d reference with %s.\n"
         pnum (string_of_t pam.pa_type));
      (
      match pam.pa_type with
```

```
              Types.MatrixUnknownSize-> (Sast.Param(pnum,indl), MatrixUnknownSize), 0
            | Types.Matrix(diml) ->
                let expr_diml = shorten_diml (eval_expr_diml indl diml p) in
                  (Param(pnum,indl), Matrix(expr_diml)), 1
            | Types.Int -> assert false
          )
  | Sast.ExtNeuron(gsid, modref, [], nname, nindl), Types.MatrixUnknownSize ->
      let {t=nt} = IntMap.find gsid p.pneurons in
          (
        match nt with
              Types.Int -> assert false
            | Types.MatrixUnknownSize ->
                ( Sast.ExtNeuron(gsid, modref, [], nname, nindl), MatrixUnknownSize), 0
            | Types.Matrix(diml) ->
                let expr_diml = shorten_diml (eval_expr_diml nindl diml p) in
                ( Sast.ExtNeuron(gsid, modref, [], nname, nindl), Matrix(expr_diml) ), 1
          )
  | Sast.NeuronRef(gsid, nname, nindl), Types.MatrixUnknownSize ->
      let {t=nt} = IntMap.find gsid p.pneurons in
          (
        match nt with
              Types.Int -> assert false
            | Types.MatrixUnknownSize ->
                ( Sast.NeuronRef(gsid, nname, nindl), MatrixUnknownSize), 0
            | Types.Matrix(diml) ->
                let expr_diml = shorten_diml (eval_expr_diml nindl diml p) in
                ( Sast.NeuronRef(gsid, nname, nindl), Matrix(expr_diml) ), 1
          )
  | e -> raise (Exception ("No resolution for " ^ (string_of_expr e)))

(* Returns synapse with expression with possibly less ambiguity and
* the number of expressions in the SAST that have been resolved. *)
let resolve_syn s p =
  let sexpr',ambigreduction = resolve_expr s.sexpr p
  in
    { sdest=s.sdest; sexpr=sexpr'; sfor=s.sfor }, ambigreduction


let rec resolve_sizes p renv =
  Printf.printf "Ambiguity = %d\n" (ambiguity renv p);
    if ambiguity renv p = 0 then p
    else
     let unr_syms, p = List.fold_left
                        (fun (li,p) pnum -> match (resolve_symbol_ref p pnum) with
                              p,NoInt     -> li, p
                            | p,Int(pnum) -> pnum::li,p )
                        ([],p) renv.unres_symbols
     in
       print_endline "After resolve_symbol_ref:";
```

```
      print_endline (IntMap.fold (fun k pam str -> (string_of_param k pam) ^ str ) p.pparams "");
    let synaps',ambigreduction =
      IntMap.fold
        (fun k s (syns,ar) ->
                     let syn,ared = resolve_syn s p in
                       IntMap.add k syn syns,(ar+ared))
        p.syn (IntMap.empty,0)
    in
     print_endline "After calling resolve_syn:";
     print_endline
       (IntMap.fold
          (fun k syn str -> (string_of_synap k syn) ^ str ) synaps' "");
    let p = { syn=synaps'; pparams=p.pparams;
              actfuns=p.actfuns; kerfuns=p.kerfuns; pneurons=p.pneurons }
    in
    let renv' = { unres_symbols = unr_syms }
    in
       if ambiguity renv p <= ambiguity renv' p && ambigreduction = 0 then
       (
          print_string (Printer.string_of_program p);
          raise ( Exception "Unable to resolve ambiguity of symbol dimensions." );
       )
       else
          resolve_sizes p renv'

let allocate_neuron_array gsid p n = match n with
    { t=Matrix(diml); v=v } ->
      let reqd_size = List.fold_left (fun a dim -> a*dim) 1 diml in
      let v' = Array.make_matrix reqd_size 2 0.0 in
        { t = Matrix(diml); v = v'; n_name=n.n_name; n_mname=n.n_mname }
  | { t=Types.Int } -> assert false;
  | { t=Types.MatrixUnknownSize } ->
      if IntMap.mem gsid p.syn = false then
        raise ( Exception (Printf.sprintf "Neuron %d is not defined by a synapse." gsid));
      let { sexpr=(expr,t) } = IntMap.find gsid p.syn in
        match t with
            Types.Matrix(diml) ->
              let reqd_size = List.fold_left (fun a dim -> a*dim) 1 diml in
              let v' = Array.make_matrix reqd_size 2 0.0 in
                { t = Types.Matrix(diml); v = v';
                  n_name=n.n_name; n_mname=n.n_mname }
          | _ -> assert false (* compiler error, all synapses have been resolved *)

let create_image_data d1 d2 d3 =  (* should allow different alignments *)
  let data' = Array3.create
                Bigarray.int8_unsigned
                Bigarray.c_layout d1 d2 d3 (* order of data is y-axis x-axis nChannels *)
  in
    Array3.fill data' 0;
```

```
    data'

let allocate_param_data pam = match pam with
    { pa_io=SpaceDelimFile(_,_); pa_type=Matrix(diml) } ->
      let data' =
        let totaldim =
          List.fold_left (fun a d -> a*d) 1 diml
        in
          Array.make totaldim 0.0
      in
      { pa_num=pam.pa_num; pa_type=pam.pa_type;
        pa_ptype=pam.pa_ptype; pa_io=pam.pa_io;
        pa_data=FloatArray(data') }

  | { pa_type=Matrix([d1; d2; d3]); pa_io=OutPpmSeq(fpat) } ->
        { pa_num=pam.pa_num; pa_type=pam.pa_type;
          pa_ptype=pam.pa_ptype;
          pa_io=OutPpmSeq(fpat); pa_data=ImageC3(create_image_data d1 d2 d3) }

  | { pa_type=Matrix([d1; d2; d3]); pa_io=InPpm(fpat,nreads) } ->
        { pa_num=pam.pa_num; pa_type=pam.pa_type;
          pa_ptype=pam.pa_ptype;
          pa_io=InPpm(fpat,nreads);
          pa_data=ImageC3(create_image_data d1 d2 d3)
        }

  | { pa_type=Types.MatrixUnknownSize; pa_num=pnum }
    -> raise( Exception (Printf.sprintf "Param␣$%d␣has␣unresolved␣size." pnum))
  | { pa_type=Types.Int } -> raise( Exception "Param␣cannot␣have␣type␣int.")
  | { pa_type=Matrix(diml) } ->
      raise( Exception "Parameter␣must␣have␣exactly␣3␣dimensions")


let program p =
   let p = resolve_sizes p (create_env p) in
     print_endline "During␣Translation2:";
     print_endline (Printer.string_of_program p);
   let neurons' = IntMap.mapi (fun key n -> allocate_neuron_array key p n) p.pneurons in
   let params' = IntMap.map allocate_param_data p.pparams in
     { syn=p.syn; pneurons=neurons'; pparams=params'; actfuns=p.actfuns; kerfuns=p.kerfuns }
```

## A.8 params.ml

```
open Set
open Ast
open Sast
open Printf
open Printer
open Ppm
```

```
open Str
open Bigarray

exception InputDeclException of string
exception Exception of string

module IntSet = Set.Make(struct type t = int let compare x y = Pervasives.compare x y end)

let openStream ( param : Sast.param_def ) =
  Printf.printf "Attempting to open %s.\n" (string_of_param param.pa_num param);
  match param with
   { pa_ptype=InParam; pa_io=SpaceDelimFile(fname,chnl) } ->
      {  pa_io = SpaceDelimFile( fname, OpenIn(open_in fname) );
         pa_num=param.pa_num; pa_type=param.pa_type;
         pa_ptype = InParam; pa_data = param.pa_data }
 | { pa_ptype=OutParam; pa_io=SpaceDelimFile(fname,chnl) } ->
      {  pa_io = SpaceDelimFile( fname, OpenOut(open_out fname) );
         pa_num=param.pa_num; pa_type=param.pa_type;
         pa_ptype = OutParam; pa_data = param.pa_data; }
 | { pa_ptype=InParam; pa_io=InPpm(fname, rtime) } ->
     let data' = Ppm.read_ppm fname in
      {  pa_io = InPpm( fname, rtime );
         pa_num=param.pa_num; pa_type=param.pa_type;
         pa_ptype = InParam; pa_data = ImageC3(data') }
 | { pa_ptype=OutParam; pa_io=OutPpmSeq(fname) } ->  (* do nothing*)
     param
 | _ -> raise( Exception "Attempted to open parameter with invalid combination." )



let rec eval_dim = function
    [] -> []
 |  Ast.Intgr(x) :: tl -> x :: (eval_dim tl)
 |  _ -> raise ( InputDeclException "error")

(* exprl should be a list of const_int_expr *)
let rec set_input_dim m (pnum, exprl) =
    let {pa_num=pa_num; pa_type=pa_type;pa_ptype=pa_ptype;
                pa_io=pa_io;pa_data=pa_data } =
      try
        Sast.IntMap.find pnum m
      with Not_found ->
        raise( Exception ("Param $"^(string_of_int pnum)^" not found in set_input_dim."))
    in let diml = eval_dim exprl in
    let totalsize = List.fold_left (fun siz i -> siz * i) 1 diml
    in let m = IntMap.add
                 pa_num
                 { pa_num=pa_num; pa_ptype=pa_ptype;
                   pa_type=Types.Matrix(diml); pa_io=pa_io;
                   pa_data=Unallocated } m
       in (m : param_def IntMap.t)
```

```

let rec find_inputs = function
    Ast.ExternalRef(param), set -> add_param( param, set)
  | Ast.Binop(expr1,_,expr2), set ->
      let set' = find_inputs (expr1, set) in
        find_inputs(expr2, set')
  | Ast.SizeOf(expr,_), set -> find_inputs (expr, set)
  | Ast.Negate(expr), set -> find_inputs (expr, set)
  | Ast.NeuronRef(_,_), set -> set (* Need to make sure indices don't include param! *)
  | Ast.ActRef(_,expr,_), set -> find_inputs( expr, set )
  | Ast.WghtRef(_,_), set -> set
  | Ast.Conv(expr,_), set -> find_inputs( expr, set )
  | Ast.Exp(expr), set -> find_inputs( expr, set )
  | Ast.Sin(expr), set -> find_inputs( expr, set )
  | Ast.Cos(expr), set -> find_inputs( expr, set )
  | Ast.Span(_,_,_), set -> set (* Param must not be in Span *)
  | _,set -> set
and add_param = function
    Ast.Param( num, _ ),set -> IntSet.add num set
  | Ast.ExtNeuron(_,_,_,_),set-> set

let rec find_outputs = function
  Ast.ExtNeuron(_,_,_,_), set -> set
| Ast.Param(num, _), set -> IntSet.add num set

(* creates an IntMap of params *)
let create (program:Ast.program) =
    let params_in = List.fold_left
                      (fun s x -> find_inputs(x.s_syn.s_expr,s))
                  IntSet.empty program.p_synap
    and params_out = List.fold_left (fun s x -> find_outputs(x.s_dest,s))
                  IntSet.empty program.p_synap
    in
    let push_param = fun pt m x ->
      let pa_io' = match pt with
          Sast.InParam ->
            if Str.string_match (Str.regexp "\\(.*\\.ppm\\):\\[\\([0-9]+\\)\\]") Sys.argv.(x) 0 then
              (
                let fname = Str.matched_group 1 Sys.argv.(x) in
                  Printf.printf "g1='%s'  g2='%s'\n" fname (Str.matched_group 2 Sys.argv.(x)) ;
                let rtime = int_of_string (Str.matched_group 2 Sys.argv.(x)) in
                  InPpm(fname,rtime)
              )
            else SpaceDelimFile(Sys.argv.(x),Closed);
        | Sast.OutParam ->
            if Str.string_match (Str.regexp "\\(.*\\.ppm\\)") Sys.argv.(x) 0 then
              (
                let fname = Str.matched_group 1 Sys.argv.(x) in
                  OutPpmSeq(fname)
              )
```

```
            else SpaceDelimFile(Sys.argv.(x),Closed);
      in
        IntMap.add x
           { pa_num=x;
             pa_type=Types.MatrixUnknownSize;
             pa_ptype=pt; pa_io= pa_io';
             pa_data = Unallocated;
           } m
    in
    let params = List.fold_left (push_param InParam) IntMap.empty (IntSet.elements params_in)
    in
      print_endline "Printing input parameters found...";
      IntMap.iter
        (fun k x -> print_endline (Printer.string_of_param k x) )
        params;
    let params = List.fold_left set_input_dim params program.p_indim
    in let params = List.fold_left
              (push_param OutParam) params (IntSet.elements params_out)
    in
       assert( IntMap.is_empty params = false );
       (*IntMap.iter (fun k x ->print x) params;*)
       params
```

## A.9 ppm.ml

```
open Printf
open Bigarray

(* from http://rosettacode.org/wiki/Read_ppm_file#OCaml *)
let read_ppm ~filename =
  Printf.printf "Trying to read %s.\n" filename;
  let ic = open_in filename in
  let line = input_line ic in
  if line <> "P6" then invalid_arg "not a P6 ppm file";
  let line = input_line ic in
  let line =
    try if line.[0] = '#'  (* skip comments *)
    then input_line ic
    else line
    with _ -> line
  in
  let width, height =
    Scanf.sscanf line "%d %d" (fun w h -> (w, h))
  in
    Printf.printf "%dx%d\n" width height;
  let line = input_line ic in
  if line <> "255" then invalid_arg "not a 8 bit depth image";
  let all_channels =
    let kind = Bigarray.int8_unsigned
    and layout = Bigarray.c_layout
```

```
    in
      Array3.create kind layout height width 3
  in
    try  (* shouldn't need this try! *)
      for y = 0 to pred height do
        for x = 0 to pred width do
          all_channels.{y,x,0} <- (input_byte ic);
          all_channels.{y,x,1} <- (input_byte ic);
          all_channels.{y,x,2} <- (input_byte ic);
        done;
      done;
      close_in ic;
      all_channels
    with End_of_file -> close_in ic; all_channels
(*,
   r_channel,
   g_channel,
   b_channel)   *)

let output_ppm ~filename
      ~(all_channels:
          (int, Bigarray.int8_unsigned_elt, Bigarray.c_layout) Array3.t ) =
  Printf.printf "\n dim=%dx%dx%d\n"
    ( Array3.dim1 all_channels )
    ( Array3.dim2 all_channels )
    ( Array3.dim3 all_channels );
  let width = Bigarray.Array3.dim2 all_channels
  and height = Bigarray.Array3.dim1 all_channels in
    Printf.printf "Image size = %dx%d\n" width height;
  let oc = open_out filename in
    Printf.fprintf oc "P6\n%d %d\n255 " width height;
    try  (* shouldn't need this try! *)
      for y = 0 to pred height do
        for x = 0 to pred width do
          (*Printf.printf "(y=%d,x=%d)..." y x; *)
          output_char oc (char_of_int all_channels.{y,x,0});
          output_char oc (char_of_int all_channels.{y,x,1});
          output_char oc (char_of_int all_channels.{y,x,2});
          (*Printf.printf "success\n"; *)
        done;
      done;
      output_char oc '\n';
      flush oc;
      close_out oc;
    with End_of_file ->
  output_char oc '\n';
  flush oc;
  close_out oc;
;;
```

## A.10 types.mli

```

type t =
    Int
(*  | Span of int (* number in span, Span(1)=Int *) *)
  | Matrix of int list
  | MatrixUnknownSize
```

## A.11 synap.ml

```
open Str
open Ast
open Sast
open Printf
open Params
open Translate1
open Translate2
open Validate
open Printer
open Eval
open Ppm
open Bigarray

exception Exception of string

let write_param pam tstep = match pam with
    { pa_io=SpaceDelimFile( fname, OpenOut(cout) ); pa_data=FloatArray(data) }->
         (*Printf.printf "$%d " pam.pa_num;
         List.iter (fun v -> print_float v; print_string "," ) (Array.to_list data );
         print_endline ""; *)
         Array.iter (fun x-> output_string cout ((string_of_float x)^" ")) data;
         output_string cout "\n";
  | { pa_io=OutPpmSeq( fpattern ); pa_data=ImageC3(data) } ->
          let (fname:string) = Printf.sprintf "../tests/images/out/temp%03d.ppm" tstep
          in
             Printf.printf "Trying to write PPM file '%s'..." fname;
             Ppm.output_ppm fname data;
             Printf.printf "success.\n"
  | _ -> ()

let update_param pam tstep =
  Printf.printf "Calling update_param %d\n" tstep;
  try
  match pam with
    { pa_io = SpaceDelimFile( fname, OpenIn(cin) ) } ->
          let line = input_line cin in
          let vals = List.rev
                         (
                         List.fold_left (fun li word ->
                                             (float_of_string word)::li )
```

```
                         [] (Str.split(Str.regexp " ") line)
                      )
          in
            Printf.printf "$%d " pam.pa_num;
            List.iter (fun v -> print_float v; print_string "," ) vals;
            print_endline "";
             ({pa_num=pam.pa_num; pa_type=pam.pa_type; pa_ptype=pam.pa_ptype;
                 pa_io=pam.pa_io;
               pa_data=FloatArray(Array.of_list vals) }, true)

  | { pa_io = InPpm( fname, maxreads ); pa_data=ImageC3(data) } ->
          if tstep > maxreads then raise End_of_file;
          Printf.printf "Trying to read PPM file '%s'..." fname;
          Array3.blit ( Ppm.read_ppm fname ) data;
          Printf.printf "success.\n";
          pam,true
        (* The output params are not updated here, they are updated when their
        * corresponding synapses are updated *)
  | { pa_io = OutPpmSeq( fpat ) } -> pam,true
  | { pa_io = InPpm( fname, maxreads ) } -> raise (Exception "InPpm must be of
      type ImageC3.")
  | { pa_io = SpaceDelimFile( fname, OpenOut(cin) ) } -> pam,true
  | { pa_io = SpaceDelimFile( fname, Closed ) } -> raise (Exception "Parameter
      file is closed!")

  with End_of_file ->
    pam,false

let rec get_step ndimsleft t =
  if ndimsleft = 0 then 1
  else
    match t with
        Types.Matrix( d :: dims ) ->
          if ndimsleft = List.length dims+1 then
            d * (get_step (ndimsleft-1) t)
          else if ndimsleft < List.length dims+1 then
            (get_step (ndimsleft-1) t)
          else
            raise (Exception "Opps hola.")
      | _ -> raise (Exception "Type of param must be matrix.")

let rec find_for_expr forvar forlist = match forlist with
    [] -> raise (Exception ("Didn't find "^ forvar ^" in for-list."))
  | ForExpr(forvar', exprd) :: forlist' ->
      if forvar=forvar' then
        ( forvar',exprd )
      else
        find_for_expr forvar forlist'
  | _ -> raise (Exception "Invalid expression found in for-list.")

let rec iter_matrix_expr f expr_dim =
  iter_matrix_expr_loop f expr_dim expr_dim []
and
```

```
    iter_matrix_expr_loop f uneval_dim expr_dim expr_indl =
  match uneval_dim with
      [] -> (f expr_indl)
    | dim_expr :: uneval_dim' ->
        for ind = 1 to dim_expr do
          (iter_matrix_expr_loop f uneval_dim' expr_dim (ind::expr_indl))
        done

(* executes f(evalenv) when evalenv has all of the forbindings *)
let rec iter_forloop f evalenv dimUneval sfor =
  match dimUneval with
      [] -> (f evalenv)
    | dim_expr :: dimUneval' ->
        let forvarl = Translate2.retrieve_forvar dim_expr in
        let forvar = match forvarl with
             [] -> raise( Exception
                            ("Blaahhk - no allowy?"))
          | [varn] -> varn
          | varn :: li ->
              raise( Exception
                       "Only a single for-variable can be in a single dimension index.")
        in
        let (forvar,span) = (find_for_expr forvar sfor) in
        let nums = Eval.nums_of_span span in
          List.iter (fun n ->
                       let evalenv' =
                         { forbindings = ( (forvar,n) :: evalenv.forbindings );
                           varbindings = evalenv.varbindings }
                       in
                         iter_forloop f evalenv' dimUneval' sfor ) nums

let iter_forloop_and_matrix_expr f evalenv indl sfor expr_diml =
  (iter_forloop
     (fun evalenv ->
        (iter_matrix_expr
           (f evalenv)
           expr_diml))
     evalenv indl sfor)

let update_synap syn p tstep =
  Printf.printf "Calling update_synap t=%d for %s\n"
    tstep (Printer.string_of_synap 0 syn);
  match syn with
    { sdest=sdest; sexpr=(sexprd,Types.Int); sfor=sfor} -> assert false
  | { sdest=sdest; sexpr=(sexprd,Types.MatrixUnknownSize); sfor=sfor} -> assert false
  | { sdest=sdest; sexpr=(sexprd,Types.Matrix(ediml)); sfor=sfor} ->
        (
          match sdest with
              Param(pnum, indl) ->
                Printf.printf("Param\n");
                let pam = (IntMap.find pnum p.pparams) in
```

```
              (
              match pam with
                  { pa_data=FloatArray(data) } ->
                    iter_forloop_and_matrix_expr
                      (fun evalenv eindl ->
                         Printf.printf "expr index=";
                         List.iter (Printf.printf "%d,") eindl;
                         Printf.printf "\n";
                         let dloc = dataloc indl p evalenv pam.pa_type eindl in
                         let valu = Eval.eval syn.sexpr eindl p evalenv in
                         Printf.printf "dloc=%d value=%.2f\n" dloc valu;
                           data.(dloc) <-  valu
                      )
                      { forbindings=[]; varbindings=StringMap.empty }
                      indl
                      sfor
                      ediml
                | { pa_data=ImageC3(data); pa_type=Types.Matrix(diml) } ->
                    iter_forloop_and_matrix_expr
                      (fun evalenv eindl ->
                         let (i1,i2,i3) = dataloc3 indl p evalenv diml eindl in
                           let pixval = int_of_float
                                          (255.*.(Eval.eval syn.sexpr eindl p evalenv))
                           in
                             data.{i1,i2,i3} <- pixval;
                      )
                      { forbindings=[]; varbindings=StringMap.empty }
                      indl
                      sfor
                      ediml;
                    Printf.printf "success!\n"
                | { pa_data=Unallocated }  -> raise (Exception "Parameter data unallocated!")
                | { pa_data=ImageC3(data)} -> raise (Exception "Image must have type of Matrix!")
              )
        | ExtNeuron( gsid, _,[],nname,nindl) ->
            Printf.printf("ExtNeuron\n");
            iter_forloop_and_matrix_expr
              (fun evalenv eindl ->
                 let newval = Eval.eval syn.sexpr eindl p evalenv in
                 let neuron = (IntMap.find gsid p.pneurons) in
                 let nloc = dataloc nindl p evalenv neuron.t eindl in
                   Printf.printf "neuron %s[(%d)] = %.3f\n" nname nloc newval;
                   print_endline (Printer.string_of_neuronvals neuron.v);
                   neuron.v.(nloc).(0) <- newval
              )
              { forbindings=[]; varbindings=StringMap.empty }
              nindl
              sfor
```

```
                    ediml

              | NeuronRef( gsid, name, nindl) ->
                  Printf.printf "NeuronRef of %d/%s\n" gsid name;
                  iter_forloop_and_matrix_expr
                    (fun evalenv eindl ->
                       Printf.printf
                         "with bindings %s\n"
                         (Printer.string_of_forbindings evalenv.forbindings);
                       let newval = Eval.eval syn.sexpr eindl p evalenv in
                       let neuron = (IntMap.find gsid p.pneurons) in
                       let nloc = dataloc nindl p evalenv neuron.t eindl in
                         Printf.printf "local neuron %s[(%d)] == %.3f\n" name nloc newval;
                         neuron.v.(nloc).(0) <- newval
                    )
                    { forbindings=[]; varbindings=StringMap.empty }
                    nindl
                    sfor
                    ediml;
                  Printf.printf("end NeuronRef\n")

              | expr -> raise(Exception
                                (Printf.sprintf
                                   "Unhandled code %s N2n9." (Printer.string_of_exprd expr )) )
          )


let rec run_step (p:Sast.prog) tstep =
  Printf.printf "#### Time step %d ####\n" tstep;
  let params',success = IntMap.fold
                          (fun k pam (m,a) ->
                               let (pam',b) = update_param pam tstep in (IntMap.add pam'.pa_num pam' m),(b && a) )
                          p.pparams (IntMap.empty, true)
  in
  let p = { syn=p.syn; pparams=params';
            actfuns=p.actfuns; kerfuns=p.kerfuns; pneurons=p.pneurons }
  in
    if success=false then ()
    else
      (
        IntMap.iter (fun k syn -> update_synap syn p tstep) p.syn;
        IntMap.iter (fun k n -> (*Printf.printf
                                   "Neuron %d [|%f,%f|]\n" k n.v.(0).(0) n.v.(1);*)
                                 for i = 0 to Array.length n.v-1 do
                                   n.v.(i).(1) <- n.v.(i).(0)
                                 done;
                         ) p.pneurons;
        IntMap.iter (fun k pam -> write_param pam tstep) params';
        run_step p (tstep+1)
```

```
      )


let run (p:Sast.prog) =
   let pparams' = IntMap.map Params.openStream p.pparams in
     run_step {syn=p.syn; pparams=pparams';
               actfuns=p.actfuns; kerfuns=p.kerfuns; pneurons=p.pneurons } 1

let _ =
  let lexbuf = Lexing.from_channel stdin in
  let p = Parser.program Scanner.token lexbuf in
  let p = Translate1.program p in
    print_endline "\n\nProgram after Translate1:";
    print_string (Printer.string_of_program p);
  let p = Translate2.program p in
    print_endline "\n\nProgram after Translate2:";
    print_string (Printer.string_of_program p);
    Validate.sast2 p;
    IntMap.iter (fun pnum pam ->
                   print_endline
                     (Printer.string_of_param pnum pam))
      p.pparams;
    ignore (run p)
```

# Appendix B

# Tests

## B.1 Test test-afun1

```

input $1 [ 5 ];

half(x) = x / 2;

$2[x] << half( $1[x] ) for x = [1:5];
```

## B.2 Test test-afun2

```

input $1 [ 5 ];

div(x;d=5) = x / d;

$2[x] << div( $1[x];d=2 ) for x = [1:5];
```

## B.3 Test test-afun3

```

input $1 [ 5 ];

half(x;d=2) = x / d;

$2[x] << half( $1[x] ) for x = [1:5];
```

## B.4 Test test-afun-chain

```

input $1 [ 5 ];

```

```
incr(x;d=1) = x + d;
half(x) = incr(x;d=1) / 2 - .5;

$2[x] << half( $1[x] ) for x = [1:5];
```

An activation function should be able to reference another activation function as long as the reference appears after the definition.

## B.5 Test test-constant-e

```
input $1[5];

$2 << $1 * e;
```

Tests the constant e and the multiplication of a matrix with a scalar.

## B.6 Test test-constant-pi

```
input $1[5];

$2 << pi * $1;
```

Tests the constant pi and the multiplication of a matrix with a scalar.

## B.7 Test test-copy5a

```

input $1 [ 5 ];

$2[x] << $1[x] for x = [1:5];
```

## B.8 Test test-copymat

```

input $1 [2,5];

$2[x,y] << $1[x,y] for x = [1:2] y=[1:5];
```

## B.9 Test test-copymat2

```

input $1 [2,2,2,2];

$2[x,y,z,w] << $1[x,y,z,w] for x = [1:2] y=[1:2] z=[1:2] w=[1:2];
```

## B.10 Test test-cos

```
input $1[5];

$2 << cos( $1 );
```

## B.11 Test test-exp

```
input $1[5];

$2 << exp( $1 );
```

## B.12 Test test-flip5a

```

input $1 [ 5 ];

$2[6-x] << $1[x] for x = [1:5];
```

## B.13 Test test-flip5b

```

input $1 [ 5 ];

$2[size($1,1)-x+1] << $1[x] for x = [1:5];
```

## B.14 Test test-flip5c

```
/* tests the use of negation inside of an index expression */
input $1 [ 5 ];

$2[size($1,1)+(-x)+1] << $1[x] for x = [1:5];
```

## B.15 Test test-kernel1d

```

input $1 [ 5 ];

kernel foo(i) = 1/(i*i+1);

$2 << $1 ** foo();
```

## B.16 Test test-matrixadd

```
input $1[5];
input $2[5];

$3 << $1+$2;
```

## B.17 Test test-matrixadd2

```
input $1[5];
input $2[5];

module m x[5], y[5] >> z[5]
{
    z[i] << x[i]+y[i] for i=[1:5];
}

m.x[i] << $1[i] for i=[1:5];
m.y[i] << $2[i] for i=[1:5];

$3[i] << m.z[i] for i=[1:5];
```

## B.18 Test test-matrixadd3

```
input $1[5];
input $2[5];

module m x[5], y[5] >> z[5]
{
    z << x+y;
}

m.x << $1;
m.y << $2;

$3 << m.z;
```

## B.19 Test test-matrixexp

```
input $1[5];

$2 << exp($1);
```

## B.20 Test test-matrixfloat-div1

```
input $1[5];

$2 << 1.0/($1+.0001);
```

## B.21 Test test-matrixfloat-div2

```
input $1[5];

$2 << $1/1;
```

## B.22 Test test-matrixmatrix-div

```
input $1[5];
input $2[5];

$3 << $1/($2+.0001);
```

## B.23 Test test-matrixnegate

```
input $1[5];

$2 << -$1;
```

## B.24 Test test-module1

```

input $1 [ 5 ];

module half in[5] >> out[5]
{
    out[i] << in[i] / 2 for i = [1:5];
}

half.in[j] << $1[j] for j = [1:5];

$2[k] << half.out[k] for k = [1:5];
```

## B.25 Test test-module1a

```

input $1[1];

module half in >> out
```

```
{
    out << in / 2;
}

half.in << $1;

$2 << half.out;
```

## B.26 Test test-scalaraddf

```
input $1[1];

$2 << $1 + .5;
```

## B.27 Test test-scalaraddi

```
input $1[1];

$2 << $1 + 1;
```

## B.28 Test test-scalararithmetic1

```
input $1[1];

$2 << 2 + ($1-1);
```

## B.29 Test test-scalararithmetic2

```
input $1[1];
input $3[1];

$2 << $3 + 1/(($1-1)/2);
```

## B.30 Test test-scalarcopy

```
input $1[1];

$2 << $1;
```

## B.31 Test test-scalarpowdiff

```
input $1[1];
input $2[1];

$3 << $1^2 - $2 * $2;
```

## B.32 Test test-sin

```
input $1[5];

$2 << sin( $1 );
```

## B.33 Test test-temporaloffset

```
input $1[1];

module foo u >> v
{
   w << u/2;
   v << u/2+w;
}

foo.u << $1;
$2 << foo.v;
```

## B.34 Test fail-afun-mat

```
/* should fail since activation functions are supposed to take and return scalars
   */
input $1 [ 5 ];

half(x;d=2) = x / d;

$2 << half( $1 );
```

## B.35 Test fail-afun-order

```
/* compiling should fail since incr is referenced before being defined */
input $1 [ 5 ];

foo(x) = incr(x) / 2;
incr(y) = y + 1;

$2[x] << foo( $1[x] ) for x = [1:5];
```

## B.36 Test fail-in-out-param

```
input $1[1];
input $2[1];

$2 << $1; /* $2 can't be both an input and output parameter */
```

## B.37 Test fail-int-div

```
/* should fail since activation functions are supposed to take and return scalars
   */
input $1 [ 5 ];

$2[i/2] << $1[i/2] for i = [2:2:10];
```

## B.38 Test fail-for-1

```
input $1[5];

/* the for variable must appear in both the source
 * and destination of the synaptic connection */
$2 << $1[i] * e for i=[1];
```

## B.39 Test fail-for-1b

```
input $1[5];
/* a for-macro must be set to a span */

$2[i] << $1[i] * e for i=[1];
```

## B.40 Test fail-module1a1

```

input $1[1];

module half in >> out
{
   out << in / 2;
}

half.in << $1;

$2 << half.in;
```

## B.41 Test fail-module1a2

```

input $1[1];

module half x >> out
{
    out << x / 2;
}

half.out << $1;

$2 << half.out;
```

## B.42 Test fail-noinput

```

$2 << 1;
```

## B.43 Test fail-timing

```
input $1[1];

module foo u >> v
{
    w << u/2;
    v << u/2+w;
}
$2 << $1;
```

## B.44 Test fail-reserved-word-t

```
input $1[5];

$2[t] << $1[t] for t=[1:5];
```

**t** is reserved for a future version of Synapse. Any use of **t** should result in an error being thrown.

## B.45 Test fail-reserved-word-end

```
input $1[5];

$2[t] << $1[t] for t=[1:end];
```

**end** is reserved for a future version of Synapse. Any use of **end** should result in an error being thrown.