NeurOn Neural Network Design Language and Compiler

1.0.0 Alpha



User Manual

Contents

* About neurOn
* Command Line Arguments
* Compilation Chain
* Neuron and Connectome files
* CNS files
* Syntax
  + Expressions
  + Groups
  + Lists
  + Assignment
* Operators
* Variables
* Expressions
* Functions
* Inline API
* Dependencies and Imports
* Makefile compilation
* Structure of a connectome file
* Encoding (internal and connectome)
* mRNA Ciphers

About neurOn

Biological neural systems are extraordinarily complex. In a SAPA system, the intention is to create a sort of hybrid logical system that behaves similarly to a biological model but built to function on modern hardware. The intent is to create programs capable of complex learning.

With rapid development in mind, it can be extremely tedious and time consuming to constantly write, re-write and edit lower level code. Thus, neurOn offers a higher level way of designing systems without worry about the underlying overhead.

The principle is similar to a hardware design language such as VHDL or Verilog. Circuits may be designed via written expressions and compiled into a final product. However, there are several major differences.

First of all, SAPA systems are dynamic. The program written and compiled will only yield an optimized program and a starting point. From there, a system is free to evolve and learn as time progresses. Secondly, the systems intended to be created are significantly more complex than an average electrical circuit. Thus, expressions are designed with extreme complexity in mind and make heavy use of loops and regional generations.

The intent of this part of the SAPA toolset makes the development of learning machines quick and (relatively) simple. The remainder of this manual contains definitions, explanations and examples to begin creating unique neural systems.

Command Line Arguments

The table below lists all accepted command line arguments. Note that options must be set in the form of -[n] or --[setting]=*value* with no spaces in an expression.

|  |  |
| --- | --- |
| CMD Option | Description |
| -d | Enable Debugging |
| -b | Enable Build (Automatic if no makefile specified) |
| -m | Searches source directory for makefile (implicitly enables build) |
| -c | Enables connectome engine compilation |
| -r | Enables runtime engine compilation |
| -v | Prints version |
| -s | Prints symbolic signature |
| -h | Prints help information |

Compilation Chain

The compilation process in neurOn is different from many other languages in the regard that it actually produces two conjugate outputs. The first is a runtime engine, and the other a ‘connectome file’, or simulation file.

The runtime engine is an executable binary with pre-built types and functionality. On startup, a built runtime engine acting a simulation environment loads a set bytecode file called a connectome, or \*.ctm, file. From here, all saved-state information from when the simulation was last interrupted can be resumed. In addition, this allows an optimized environment to load compatible network designs.

At the first stage is the pre-compiler. Pre-compiler statements are implemented following the '?' character. Currently there exist only two such statements: *import* and *psuedo*.

Import takes one or more arguments. These arguments are additional \*.nrn source files to be imported. Multiple declarations of the same source file are ignored to prevent circular dependency loops. The \*.nrn extension is added automatically to the argument, and so should be left off. Adding multiple files will automatically import those as well. Note that unless using the -l command line option, these files will be pulled relative to the calling directory, as will all other file operations.

Psuedo, similar to *typedef* or *define* in C languages, is simply used to create a macro for another word. For instance, the line *?psuedo signal:0 eHi* will replace all instances of eHi with signal:0.

Beyond this, the pre-processor pieces together all source files into one large source file, as well as cleaning up additional whitespace and removing comments.

Immediately following per-compilation begins the lexical analysis. At this stage, a stream of tokens containing meta data for each identifiable term is generated. These tokens identify terms by keywords, primitive type (such as string, char, int and float...) and special operators such as logical and arithmetic expressions.

This token stream is then passed to the semantic analysis. Here, the token stream is scanned and a dependency profile is generated, detailing existing objects in scope-form. This is the final stage in the code analysis. The result is a bytecode file capable of being converted into the desired language, or generating neural design files.

Neuron and Connectome Files

Connectome files project files that detail information on a particular build in a single location. But what neurOn and connectome files? While they both contain information pertaining to the design of a system, they are very different.

A neurOn files is a human-readable source code file. The language outlined in this manual explains its syntax and use. However, it would be very resource consuming for the simulation enviornment to attempt to execute this code continuously. Therefore, the connectome file is generated from source neurOn files.

The connectome file is a sort of save-state file. It doesn not detail WHAT a system is comprised of, only HOW it is comprised. For instance, the connectome file has no idea what methods a particular object may have. What is does know, however, is what those objects' values were at the end of the last simulation, and how the paths between structures looked like.

A connectome file is used to not only begin a simulation, but also changes as the simulation changes, so that it may be resumed at a later date. However, they can become enourmously complex as they explicitly record every single object and data instance, and is almost entirely composed of hexadecimal values.

CNS files

A CNS file is the project file used by the compiler to generate connectomes and runtime engines. Here, compilation settings can be set, and the required neurOn source files are identified.

There are several system settings that may be set from this file. The CNS file is interpreted separately from the \*.nrn file, and so does not follow the neurOn syntax.

Below is an example CNS files:

**#test CNS file**

**#SAPA project file**

**#directory of**

**# = SET**

**# ^ IMPORT**

**^ src**

**#compilation settings**

**= evolution false**

**= optimize false**

**= debug true**

**= title dev**

**= version 1.0.0**

**= language c++**

**= makeRTE false**

**= makeCTM false**

**= prsvSymb false**

**#sim settings**

**= RSP -70#resting potential**

**= TDF 1.0#time dilation factor**

**= NGEN true#neuro genesis**

**= NMAX 500#max allowed neurons. 0 for infinite**

**= NTRAN 2#multiple of 8 neurotransmitter slots. Must have at least 1. Max 8 for 64 bit system, 4 for 32**

*example CNS file*

Noteably, there are several settings to be modified, each following the '=' operator. The first argument defined is the setting name (case sensitive). The next is the assignment value.

Here's a quick explanation on each of the current CNS settings.

Evolution: An experimental option for self modifying binaries. This is not currently implemented, and may be removed/re-thought. A future implementation may be for the simulation to pause occasionally and self modify its neural pathways via genetic algorithms.

Optimize: Runs additional scans to make a more complex, but more efficient compiled system.

Debug: If enabled, the compiled simulation environment will print diagnostic information.

Title: Will be compiled into retrievable project information, and will be set as the heading of a terminal when executed.

Version: Will be compiled into retrievable project information, and will be appended to title in header.

Language: This defines the language that the simulation enviornment source code is compiled to. The currently supported languages are C and Java.

MakeRTE: When this is true, the compiler will generate a simulation environment source file.

MakeCTE: When this is true, the compiler will generate a connectome file.

PrsvSymb: This option is not currently implemented

RSP: This is simulation resting potential.

TDF: The time dilation factor. This controls the rate of time as perceived by the simulation. Adjusting this may result in unstable performance.

NGEN: When true will allow for the simulation to prune unneeded neurons, and also spawn new cells.

NMAX: The maximum of active cells at one time.

NTRAN: Number of bytes allocated for neurotransmitter models. By default, this is one byte, allowing for 8 neurotransmitters to be defined. This number is limited to 4 on a 32 bit system, and 8 on a 64 bit system. This number must be at least 1.

Syntax

This language is inherently structural. All core operations, such as the *update*, *receive*, and *fire* methods of cells, have pre-defined functionality. However, these may be overridden on a cell-by-cell basis, or a global implementation may be used, if applicable.

Most of the work is to be done inside a defined body, such as in most common markup languages. For example, a definition of a cell would be:

**new cell[**

**name=”democell”;**

**new float[name=”charge”; value=20;]**

**override**

**]**

Operators

|  |  |
| --- | --- |
| Character | Description |
| # | Comment line, or until next ‘#’ on same line |
| #\* / \*# | Multiline comment |
| ! & | !& !| || !| | Logical operators: not, and, or, nand, nor, xor, xnor |
| + - / % \* √ ^ | Addition, Subtract, Divide, modulo, root, power |
| $ | Pre-Compiler option |
|  |  |
|  |  |

Variables

Variables in a Sapa project tend to be very limited, with the few directly available data types designed to optimize processing, and minimize memory usage.

Macros

Certain macros are pre-registered with the compiler. These include:

* True : 1
* False : 0
* SgSz : Signal Size (Defined in project file)
* OS : LINUX or WINDOWS
* BM : Bitmode (32 or 64)
* LANG : Language (Only available through simulation environment)
* VERSION : Project defined version
* ENCODEVRS : mRNA encoder version
* SIGNATURE : Data structure signature

Expressions

Functions

There are a small handful of built-in functions available. These keywords consist of declarations, IO and debugging methods, as well as common operations such as loops.

Functions can be called via the formation *functionName(param, param…)*. New methods can of course be created in a similar fashion. The *new, const* and *override* attributes are applicable. The *new* operator is implicit, and the *override* method will commonly be used when dealing with interface methods, such as ‘update’.

loop:

*loop(initial condition, ending condition, post-cycle operation)*

*loop(collection, instance symbol)*

In the first overload of the function, the subsequent function body will iterate until an ending condition is met, or the cycle is broken. In the second, the loop will cycle until each element in a collection (such as an array or synapse object).

Inline API

Dependencies and Imports

Makefile Compilation

Makefiles are a loosely supported feature. Template makefiles are supplied for supported languages in the **support/<language>/** directory. These templates may be modified if desired, but otherwise the compiler will fill in empty fields automatically when binary compilation is specified.

However, template makefiles will only be used if an existing makefile does not exist. If a makefile already exists inside the target project directory, this will be used instead. The compiler will not call a compiler directly, and thus requires that the appropriate compilers are installed on the working system, as well as a *make* program.

By default, GNU compilers will be used.

Structure of a Connectome File

Encoding (Internal and Connectome)

In order to simply the way in which information is processed and stored, encoded strings are used to explain the way information is to be structured, and how information is to be processed. This bytecode format uses hexadecimal command codes and chunk size references. Listed at the end of this segment is the particular command code set used by the internal compilation units as well as the simulation environments themselves. In this way, data may be easily serialized and searched,

The reason this method is used in the internal compilation chain is to simplify the process of extending the language, although the actual information stored may be too complex to read manually. By keeping a particular grammar, new additions and renovations to the language is simple while the engine is in a volatile development stage.

The downside of this, of course, is a significant hit to efficiency. To remedy this to some degree, objects searchable in the information registry are still separate entities, although their content is in a single encoded format, sans particular attributes used by the system.

Grammatical rules used by this system are actually fairly simple. At first, a command code ranging from 0x00 to 0xFF is followed by a value specifying the number of parameters. These parameters consist of three values. First is a value identifying the type of the following data structure. Next is the size in number of characters of the information set. And lastly is the information itself. For instance, the line of code

**float x = 28;**

would have the bytecode sequence

**[def] [float] [1] [x] [end] [assign] [float] [1] [x] [int] [2] [1C] [end]**

Of course, with the plain text replaced with hex values. In the case of '1C', this would be stored as plain-text hex as to allow values greater than 0xFF.

Each set of opcodes and identifiers will modify the way proceeding sets of data are interpreted. Components that request a set of arguments behave as singular objects, included with said arguments. Thus nested methods and encapsulations do not need to factor into the size information of lower-level bodies.

Mathematical functions are converted into post-fix notation during encoding, and are prefaced with a Math value identifier followed by a size parameter of the entire parsed section.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 0x0 | End Of File | 0X14 | Mod | 0X28 | Outside <> |
| 0x1 | End Of Statement (;) | 0X15 | Pow | 0X29 |  |
| 0x2 | Start Body [ | 0X16 | Root | 0X2A |  |
| 0x3 | End Body ] | 0X17 |  | 0X2B |  |
| 0x4 | Start Group ( | 0X18 |  | 0X2C |  |
| 0x5 | End Group ) | 0X19 |  | 0X2D |  |
| 0x6 | Start Bracket | 0X1A | And | 0X2E |  |
| 0x7 | End Bracket | 0X1B | Or | 0X2F |  |
| 0x8 | Split Item (,) | 0X1C | Not | 0X30 | Cell (identifier) |
| 0x9 | Member (.) | 0X1D | Nand | 0X31 | Group(identifier) |
| 0XA | Index(:) | 0X1E | Nor | 0X32 | Float(identifier) |
| 0XB |  | 0X1F | Xor | 0X33 | Int(identifier) |
| 0XC |  | 0X20 | XNor | 0X34 | Bool(identifier) |
| 0XD |  | 0X21 | Equ | 0X35 | Char(identifier) |
| 0XE |  | 0X22 | NEqu | 0X36 | String(identifier) |
| 0XF |  | 0X23 | Gtr | 0X37 | Fiber(identifier) |
| 0X10 | Add | 0X24 | Lss | 0X38 | Expression (identifier) |
| 0X11 | Sub | 0X25 | GtrEqu | 0X39 | Synapse (identifier) |
| 0X12 | Div | 0X26 | LssEqu | 0X3A | Interface (identifier) |
| 0X13 | Mult | 0X27 | Within >< | 0X3B | Struct (identifier) |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 0x3C | Exprss(identifier) | 0x52 | Expression(keyword) | 0x68 |  |
| 0X3D | Synapse(identifier) | 0x53 | Synapse(keyword) | 0x69 |  |
| 0X3E | DeviceOut (Identifier) | 0x54 | Interface (keyword) | 0x6A | Rand |
| 0X3F | DeviceIn (Identifier) | 0x55 | Struct (keyword) | 0x6B | Display |
| 0X40 | Constant | 0x56 |  | 0x6C | Startlog |
| 0X41 | Override | 0x57 |  | 0x6D | Wlog |
| 0X42 | New | 0x58 |  | 0x6E | Clog |
| 0X43 | Delete | 0x59 |  | 0x6F | Usin |
| 0X44 | Persistant | 0x5A | Set | 0x70 | Bind |
| 0X45 |  | 0x5B | AddSet | 0x71 | Copy |
| 0X46 |  | 0x5C | SubSet | 0x72 | Spawn |
| 0X47 |  | 0x5D | MultSet | 0x73 | Kill |
| 0X48 |  | 0x5E | DivSet | 0x74 | Unbind |
| 0X49 |  | 0x5F | Increment | 0x75 | Guass |
| 0X4A | Cell (keyword) | 0x60 | Decrement | 0x76 |  |
| 0X4B | Group(keyword) | 0x61 |  | 0x77 |  |
| 0X4C | Float(keyword) | 0x62 |  | 0x78 |  |
| 0X4D | Int(keyword) | 0x63 |  | 0x79 |  |
| 0X4E | Bool(keyword) | 0x64 |  | 0x7A | Stn (serial to neural) |
| 0X4F | Char(keyword) | 0x65 |  | 0x7B | Nts (Neural to serial) |
| 0x50 | String(keyword) | 0x66 |  | 0x7C | Usb |
| 0x51 | Fiber(keyword) | 0x67 |  | 0x7D | Fncgen |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 0x7E |  | 0x94 |  | 0xAA |  |
| 0x7F |  | 0x95 |  | 0xAB |  |
| 0x80 | And (keyword) | 0x96 |  | 0xAC |  |
| 0x81 | Or(keyword) | 0x97 |  | 0xAD |  |
| 0x82 | Not(keyword) | 0x98 |  | 0xAE |  |
| 0x83 | Nand(keyword) | 0x99 |  | 0xAF |  |
| 0x84 | Nor(keyword) | 0x9A |  | 0xB0 | Math (internal) |
| 0x85 | Xor(keyword) | 0x9B |  | 0xB1 |  |
| 0x86 | Xnor(keyword) | 0x9C |  | 0xB2 |  |
| 0x87 |  | 0x9D |  | 0xB3 |  |
| 0x88 |  | 0x9E |  | 0xB4 |  |
| 0x89 |  | 0x9F |  | 0xB5 |  |
| 0x8A |  | 0xA0 | jump | 0xB6 |  |
| 0x8B |  | 0xA1 | break | 0xB7 |  |
| 0x8C |  | 0xA2 | goto | 0xB8 |  |
| 0x8D |  | 0xA3 |  | 0xB9 |  |
| 0x8E |  | 0xA4 |  | 0xBA |  |
| 0x8F |  | 0xA5 |  | 0xBB |  |
| 0x90 | Grid | 0xA6 |  | 0xBC |  |
| 0x91 | Matrix | 0xA7 |  | 0xBD |  |
| 0x92 | Ring | 0xA8 |  | 0xBE |  |
| 0x93 | Line | 0xA9 |  | 0xBF |  |

mRNA Ciphers

One of the most powerful features, granted also one of the most complex, is the mRNA cipher file. This is a file placed in the language support directory (**support/<language>/**) that provides information on how to convert the internal encoded language into the compliable language of choice.