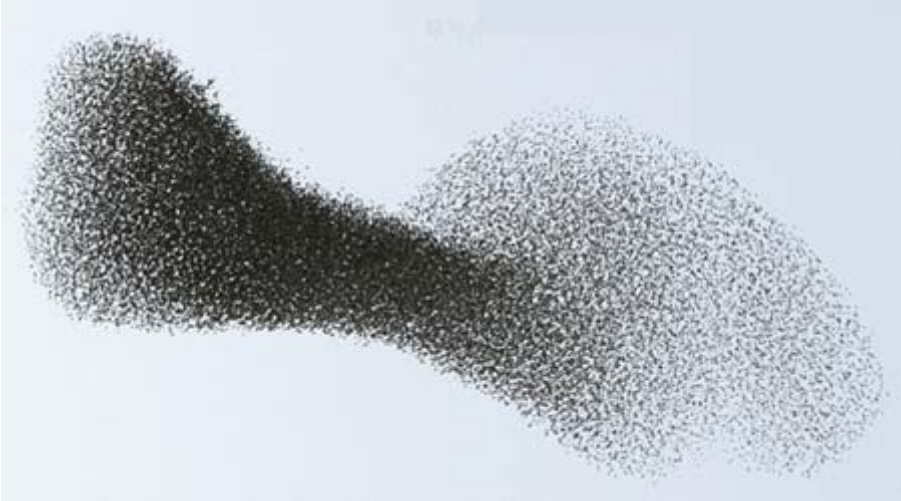
Swarm

Final Report



A Cellular Programming Language

Team Epidemic

Rajesh Ramakrishnan rr2318@columbia.edu
Jason Gluckman jbg2113@columbia.edu
Thomas Chau tc2165@columbia.edu
Greg Bramble gmb2106@columbia.edu

Programming Languages and Translators, Fall 2007: Prof. Stephen Edwards.

Table of Contents

White Paper

Abstract

Swarm is a functional dataflow language aimed at building simulations. Its programs are represented by directed graphs of cells. Cells, the basic functional unit, perform computation by responding to particular sets of stimuli and propagate reactions to their neighbors. To achieve a more intricate machine, a programmer must compose primitive cells into more complex ones, which in turn may be composed into yet larger units.

Motivation

Everything in the world behaves in a cellular manner, participating in vast interactions in which units feel each others' influences and emit their own. For example, by universally speaking the language of electromagnetism, atoms compose many interesting molecules. These complexes then compose themselves into larger structures, giving rise to cells. The cells, on the next order of magnitude, act in concert and unite themselves by circulatory networks. Finally, humans, too, interact in a broad social framework and construct organizations to handle mass action. While much reductionist work has been done to probe basic properties of these systems' constituents, important applications in biology and social science rely on understanding aggregate behavior in general. Fortunately, the current trends in hardware are beginning to meet the needs of such systems where complexity arises from the synergy of many parallel elements.

The days of the single processor are becoming history. They were characterized by programs which were essentially lists of instructions to be processed sequentially. Nearly all high-level languages in existence were written for and conceptually constrained by this paradigm. Even object-oriented programming, whose idea was to group instructions as more distinct entities, is firmly implanted in the original paradigm. While objects maintain their own states, they are nonetheless are filled with functions that must execute and return before the "main" function can continue. The OO paradigm more resembles a guideline for how to organize folders than a break from the procedural style that preceded it. As a consequence, we face a software crisis in which traditional languages have failed to capture parallel computation. We ask ourselves: how can we break away and write code to exploit new multiple processor technology?

Goals

The goal of Swarm is to allow a programmer to *explore emergent phenomena conveniently*. But even more importantly, Swarm aims to establish a platform on which a *programmer can write code that parallelizes naturally*.

Swarm's style is declarative, capturing the computation in a manner similar to that of functional languages. Its programs, rather than executing operations imperatively, define the juxtaposition of data and functionality explicitly. Swarm is a metalanguage whose operators perform high-level manipulations of the superstructure of a network of functional units. Swarm operators, when executed, link together these functional units, the *cells*, which provide conventional functionality.

Language Tutorial

What is a swarm?

> *If you have ever observed ants marching in and out of a nest, you might have been reminded of a highway buzzing with traffic. To Iain D. Couzin, such a comparison is a cruel insult — to the ants.*
> *Americans spend a 3.7 billion hours a year in congested traffic. But you will never see ants stuck in gridlock.*
> *Army ants, which Dr. Couzin has spent much time observing in Panama, are particularly good at moving in swarms. If they have to travel over a depression in the ground, they erect bridges so that they can proceed as quickly as possible.*
> *"They build the bridges with their living bodies," said Dr. Couzin, a mathematical biologist at Princeton University and the University of Oxford. "They build them up if they're required, and they dissolve if they're not being used."*
> *The reason may be that the ants have had a lot more time to adapt to living in big groups. "We haven't evolved in the societies we currently live in," Dr. Couzin said.*
> *By studying army ants — as well as birds, fish, locusts and other swarming animals — Dr. Couzin and his colleagues are starting to discover simple rules that allow swarms to work so well. Those rules allow thousands of relatively simple animals to form a collective brain able to make decisions and move like a single organism.*
> **from the *New York Times*, November 13, 2007**

What is Swarm?

Swarm is a functional programming language that attempts to capture the idea from the above article. Swarm consists of three basic ideas whose combination results in very powerful Swarm simulations. These three ideas are cells, facets, and binds.

To understand these three basic ideas, which is the goal of this tutorial, we will look at a simple program that prints "Hello World" to output:

```
[STRLIT:"Hello world!"->SOUT]:MAIN;
```

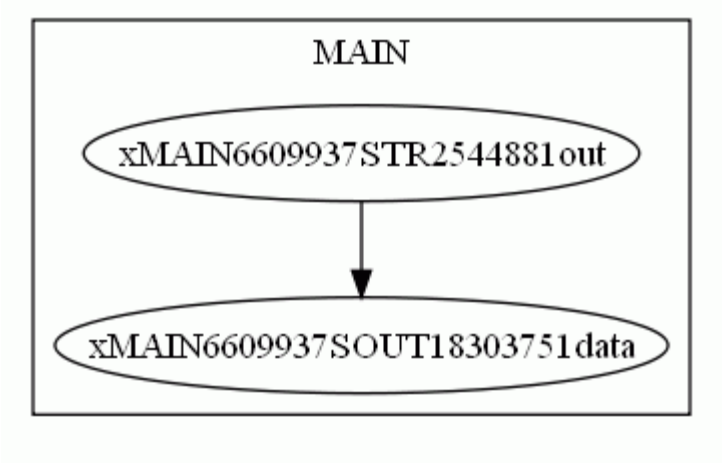
The above code constructs a STR cell (which is a built-in cell) with the value "Hello world!" The out facet of the STR cell is bound to the IN facet of the SOUT (also predefined) cell. The entire cell structure is specified between the brackets and it is defined as the MAIN cell. The MAIN cell is a special cell which the compiler recognizes as the place to begin execution.

Tokenizing:

```
[ /* Start cell definition */
  STRLIT /* Invoke a STRLIT cell */
    : /* Initialized [given a label] */
    "Hello world!" /* STRLIT cell internalizes the string */
  -> /* Cross-bind – pass the output to ... */
  SOUT /* Standard output cell (emits to the command line) */
] /* end of cell definition */
```

```
: MAIN; /* Define the cell definition [..] as MAIN
```

One of the most interesting features in Swarm is that each program run generates a graph file that shows data propagation for that program. Using the DOT language [part of the GraphViz package], graphs are synthesized from the user-defined cells. The generated file for the above "Hello World" program [debugging symbols inserted intentionally] is shown below:



Each cell in Swarm exists in its own domain. In this case, there is no need to entangle the behavior of a generalized string of characters with the behaviors of the standard output. This is further emphasized in the diagram, as the predefined SOUT and STR cells do not expose their internal structure.

The MAIN cell is fully aware of the it's internal binds. Each cell instance, as shown in the diagram, is only aware of its parent. Decomposing the mess, the number "6609937" is the cell's handle to the binding [the generalize connection] on its specific output or input facet. The facets are specified in the debugging symbol at the very end. STR exposes an ".out" output facet, while SOUT exposes a ".data" input facet.

A Full Example

Users can define their own cells. For example:

```
[(.in)->SOUT]:HELLO;
[STRLIT:"Hello world!"->HELLO]:MAIN;
```

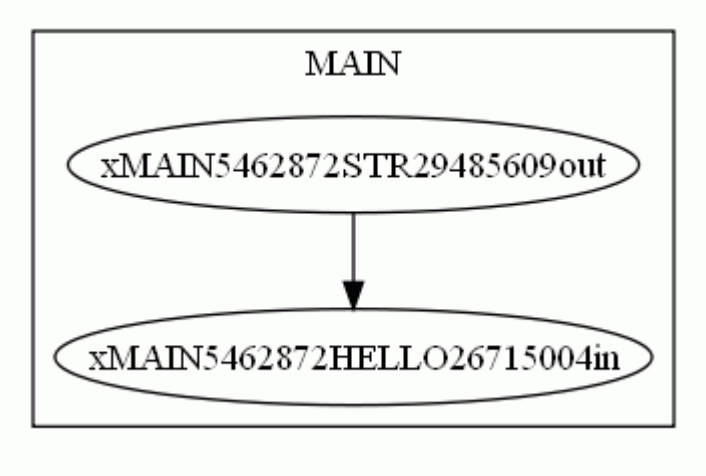
The HELLO cell is a cell defined in the code itself. When developing complex applications, you will most definitely want to create many user-defined cells like HELLO [each cell would encapsulate a self-contained process].

In order to execute the program, one would run

```
$ cat hello.swm
[(.in)->SOUT]:HELLO;
[STR:"Hello world!"->HELLO]:MAIN;
```

```
$ java Swarm hello.swm
Hello world!
```

To see the graphical output, the file “swarm.graph” has the graphical output. Running dot on the generated output yields the following:



As you can see, while creating a user-defined cell made life simpler for the programmer, Swarm handled both cases in the same way. To Swarm, internal and user-defined cells, for the most part, are equivalent. There are a few syntactical differences related to constants, but that will be covered later.

The diagram highlights a key point of Swarm: a cell should not worry about the implementation details of its sub-cells.

Reference Manual

Introduction

Model of Computation

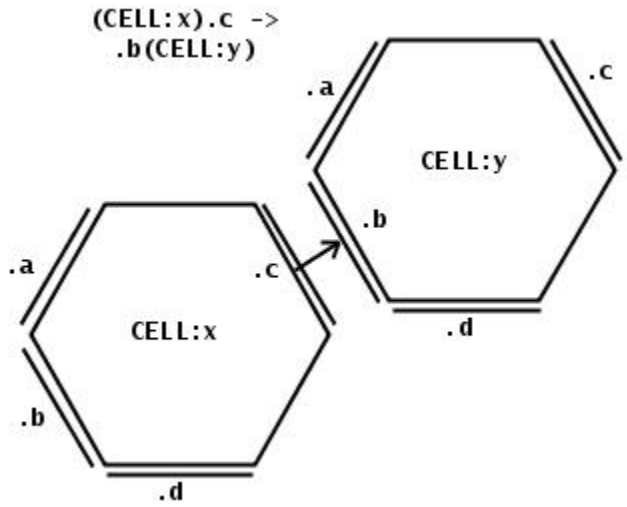


Figure 1: A hexagonal depiction of two cells and a bind.

Programs in Swarm are the structural definitions of cellular lattices. Figure 1A depicts a very simple lattice consisting of two cells and the snippet of code that describes it.

In Figure 1, assuming that the type named "CELL" is defined, the language construct "CELL:x" instantiates a new cell of that type and labels it with the symbol 'x'. One of its facets (named .c) is then bound to some facet (named .b) of another cell of the same type named 'y'.

When Swarm programs are run, a lattice of cells is constructed and some computation may begin when any of the cells receive stimuli (pieces of data emitted by another cell). All data is cellular in nature; the idea of "primitive types" is also considered as cellular. The initial stimulus may come from a MAIN cell that is automatically started, similar to main functions in C, or from the user via a GUI or standard input. Stimulation occurs when data is transferred from one cell to another via their facets.

The Cell

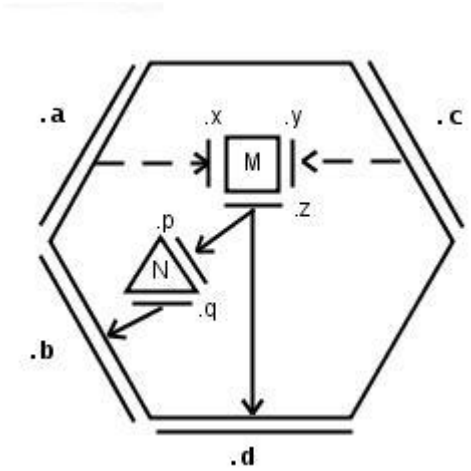


Figure 2: A Cell [and internal substructure]

The most important concept is that of the cell. It is the primary functional unit that may be bound to other units and responds to environmental stimuli by performing an internal computation and then secreting a response. The structure of all programs in Swarm takes the form of directed graphs. These graphs may be manipulated by cells themselves, which create, destroy, bind, and unbind each other. Primitive types such as numbers and text are special cells which propagate the value of their data to their neighbors. To facilitate the construction of these graphs (which we shall refer to as *lattices*), cells' connectivity is declared in terms of links between facets (defined binding sites). Note that while these facets define interfaces between units, they are totally unlike Java "interfaces" in that the possible ways to stimulate the facet's binding sites are not enforced by the compiler; in addition, the facets may be stimulated by or observed by arbitrarily many nearby cells. After a cell performs a computation, its output values are propagated automatically through facets. Cells have a recursive relationship with each other in that cells may be composed to define composite cells. In such derived cells, the programmer specifies how sub-cells interface each other as well as how they interface with the membrane of the enclosing cell.

The Facet

Facets are points of control of the data flow. Binding between facets is not checked at compilation: there are no strict "binding types". However, a cell membrane's binding sites may filter the stimuli they receive, deciding whether to permit data inside the cell based on the data's value, properties, or type (see TYPE in the section Special Semantics). The cells are only stimulated to compute when the proper set of stimuli impinge on their binding sites. The stimulating data then "binds and waits", attaching to the facet until the facet is ready to internalize it. When the facet is ready, all values diffused into the cell and fed to the cell's interior lattice of sub-cells, which will perform a computation and possibly propagate values out its facets as output. More details on how a lattice executes and propagates the computation is explained in the section "Execution Flow".

Binding

A binding defines a flow of data from facet to facet. In general, arbitrarily many facets may bind to any one facet; also, facets may be used bidirectionally -- a facet can be specified to be used for emitting as well as receiving data. A programmer *positively defines* bindings between cells, such that in the case of undefined output flows, the values emitted are unused. That is, if some cell defines behavior for both input and output to one of its facets, the programmer can bind another cell to it unidirectionally; **the availability of an opposite direction does not force the programmer to utilize both. See the section on binding operators for implementation details.**

Lexical Conventions

Swarm has no keywords. Whitespaceiscompletelyignored!

Comments

The characters `/*` introduce a comment, which terminates with the characters `*/`. Comments do not nest, and they do not occur within string literals.

Identifiers

There are three types of identifiers.

1. A CELLTYPE, which serves as a label for the structural definition of a cell, is a sequence of one or more uppercase letters.

2. An INSTANCE is a sequence of one or more lowercase letters or a sequence of one or more digits, but not both. An INSTANCE labels a particular concrete instance of some cell.

3. A FACET is a '.' followed by a sequence of one or more lowercase characters. When succeeded by a CELLTYPE or INSTANCE in parens (e.g. .input(MYCELL), the FACET is considered to be contextualized to that cell. Likewise, the FACET may be preceded (e.g. MYCELL.output). Allowance of both techniques allows for chaining of bind operations. When alone, the FACET is contextualized to the nearest enclosing named cell definition.

Special Semantics

Swarm provides certain special built-in cells. They have a special-case semantic regarding the identifier to the right of the colon. They use these identifiers as qualifiers or as literals to complete the cell's behavior. For example, the STR cell type needs a string literal in order to be instantiated and to have a value to propagate.

Primitive data cells:

Primitive data type cells have one single input facet, (.to), and one output facet, (.out). and can be instantiated using the : operator. The input facet, (.to), converts an input passed to that facet into that data type. For example, `[STR:"42.0"->INT->SOUT]` will convert the STR "42" into the FLT 42.0, then print it out. The output facet, (.out), outputs this cell.

INT

INT is the equivalent of "int" in C.
Example instantiation: `INT:″-5″`

STR

STR is the equivalent of "String" in Java. A BOOL converted to a STR will appear as "0" or "1".
Example instantiation: `STR:"Hello Swarm!\n"`

Main Cell:

MAIN

MAIN is a very special purpose cell which identifies to the compiler where the actual program instruction sequence lies. There must be exactly one MAIN cell in any program, and MAIN cells in included files will be ignored. Main does not have any input or output facets.
Example: `[INT:42->STR->SOUT]:MAIN` defines a program that prints the STR "42" to STDOUT.

Meaning of Identifiers

The CELLTYPE, or cell, is a functional unit that can be bound to other functional units by one or more Facets that belong to it. Its structure is encoded between square brackets.

The INSTANCE is an identifier that corresponds to a particular concrete instance of one type of cell. If it is addressed in multiple statements or in multiple bindings, the operators will act upon the same object. When a cell is not labeled in this manner, every use of its type in a binding relationship will assume a distinct and nameless new cell of that type.

The FACET is an identifier which symbolically references a particular named facet of either the contextualizing cell or of the enclosing cell. A facet is contextualized by a following cell identifier enclosed in parentheses or by a a preceding cell identifier optionally enclosed in parentheses. Groups of facets can be referenced and contextualized; in such cases, the facets in the group must be enclosed in parentheses and delimited by commas. The group is contextualized in the same manner as individual facets relative to a cell identifier.

Generally: prefixing enforces enclosing of the cell identifier in parentheses. Cells may be surrounded by any combination of individuals or groups.

Non-exhaustive Set of Examples of Valid forms:

```
.facet(CELL)  singular prefixed facet
CELL.facet    singular postfixed facet
```

```
(.faceta, .facetb)(CELL)(.facetc, .facetd)  postfixed and
prefixed groups of facets
```

The facets are notated in both prefix and postfix forms in order to facilitate "chains" of binding operations. See section on **Binding Operators** for an explanation of why prefix and postfix exist and how they are used.

Predefined Cells

Swarm includes several cells that are part of the core library. The descriptions are below:

Math cells:

ADD

ADD takes in n > 0 INT arguments and adds them. The individual input facets of ADD cannot be accessed individually, due to the nature of the cell.

```
<(*) ADD (.out)>
```
*: any number greater than 0 of INT inputs
out: type INT

MULT

MULT takes in n arguments and multiplies them. MULT will always output type INT. The individual input facets of MULT cannot be accessed individually, due to the nature of the cell.

```
<(*) MULT (.out)>
```
*: any number greater than 0 of INT inputs
out: an INT

MOD

MOD takes in two arguments and computes the modulus, where the first argument is the dividend and the second argument is the divisor. MOD will always output type INT.

```
<(.a, .b) MOD (.out)>
```
a: an INT
b: an INT
out: an INT

Comparison Cells:

GREATER_THAN

GREATER_THAN takes in two INT arguments and compares them. GREATER_THAN outputs 1 if the left input is greater than the right, and NIL otherwise.

```
<(.left, .right) GREATER_THAN (.output)>
```
left: an INT
right: an INT
output: an INT

LESS_THAN

LESS_THAN takes in two INT arguments and compares them. LESS_THAN outputs 1 if the left input is less than the right, and NIL otherwise.

```
<(.left, .right) LESS_THAN (.output)>
```
left: an INT
right: an INT
output: an INT

EQUALS

EQUALS takes in two INT arguments and compares them. EQUALS outputs 1 if the two INT's are equal and 0 otherwise.

```
<(.left, .right) EQUALS (.output)>
```
left: an INT
right: an INT
output: an INT

Input / Output Cells:

GUI

GUI takes in an argument, either 0 or 1, determining whether the element should be on or off (black for 1, white for 0). GUI must also be given a name which is its location on the 15 by 15 grid.

```
<(.in) GUI:"x_loc,y_loc">
```
in: 0 or 1

SOUT

SOUT takes in an argument and prints it out onto standard output. If the argument is not an STR it will be converted to an STR before printing.

```
<(.data) SOUT>
```

data: an INT or STR

Derived Cells

Derived cells arise from compositions of fundamental cells and other derived cells.

Derived cells are declared as the binding of other cells. The structural definition is enclosed in square brackets and, if the cell is named, followed by a cell type name in all capital letters. All uncontextualized facets in are assumed to be owned by the innermost enclosing named cell. There are two types of derived cells:

1. *Named cells* have a cell type identifier (the name) and named facets. The structural definition (enclosed in brackets) specifies how facets and any internal cells are linked.

2. *Anonymous cells* cannot have named facets. They may be declared within an enclosing cell definition. They also have a special semantics triggered by any input whatsoever; when stimulated, they temporarily alter the enclosing cell's binding according to the lattice defined within the anonymous cell's braces. The reconfiguration endures only for the current computation.

Conversions

Conversion between types of literals (or casting) is accomplished through use of the .to facet, which all built-in data types have.

Example: `INT:5->.to(STR)`
This example converts the int literal 5 to the string "5".

Declarations

Literals are defined through the special semantics of the INT and STR cells. There is no concept of "Variable" in the C-sense. The analogue of the variable is the facet, which take in data and are specified at the outset.

Statements

Cellular lattice definitions consist of a list of statements separated by semicolons, or optionally terminated by semicolons. Each statement is one chain of facets or cells bound to other facets or cells. The binds between blocks define the nature of connections. To resolve more complicated loops [cells which take in multiple inputs in different chains], multiple statements can refer to the same objects. For example:

```
.a -> CELL:b -> .c; .c -> b;
```

The two statements above refer to the same cell instance, labeled as `b`

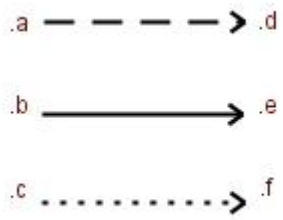
Binding Operators

The binding operators allow cells to be linked together through their facets. There are two types of binding, direct binding and cross binding.

=> [Direct Bind]

This binds groups of facets element-wise, such that the nth facet in one group binds the nth facet in another group. For example:

```
(INT:3, INT:4) =>  (.a, .b)(HYPOTENUSE)
```



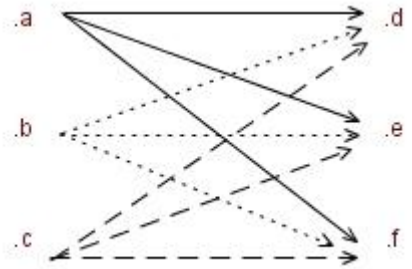
This binds the primitive cells for integers 3 and 4 to the .a and .b facets, respectively.

-> [Cross Bind]

This binds elements in groups of facets in a cross-product. In graph theoretical terms, it produces the complete bipartite graph $K_{n,m}$ where n and m are the number of facets in the first and second groups respectively. Binding two cells directly will perform the equivalent operation on their entire facet sets. For example:

```
(.a,.b,.c) -> (.d, .e, .f)
```

produces:



Chaining

To permit succinct expression of multiple relationships, the binding operators may be combined into chains, in which the postfixed facets of a cell are bound to the prefixed facets of another cell. A list of several statements, each containing a chain, constructs a full cell definition and describes a cellular structure. An example of a chain:

```
.a(A).b -> .c(D).e -> (.f(G).h, .i(J).k) => L
```

Execution Flow

A program is defined as the contents of the MAIN cell. Programs execute as the propagation of data, which flows along the paths positively declared by the programmer. However, not all data will be propagated along all paths in sync. Therefore, a facet requires that all required inputs impinge upon it before it triggers a computation within the cell. For example, if a facet requires three integers, and a programmer had three cells of differing complexity feed it, the facet would wait until all three feeders had completely sent their values before initiating a calculation. The earliest values simply "bind and wait," to be internalized only after the last value arrives. Once the facet triggers the cell, the outputs of that computation will be released simultaneously (if it branches inside and flows to different output facets); outbound values will also have the "bind and wait" property. Discrepancies in cells' "speed" are spooled in this way. Conceptually, facets represent the idea of data dependency by naturally expressing the points at which various task threads need to wait upon each other before progressing.

Because cells will only compute when stimulated with valid inputs, program "flow" is controlled by whether cells propagate outputs or not. An example of this can be seen in the TYPE cell. Entire regions of a cell may not be activated. A cell's flow may also be reconfigured by the activation of anonymous cells that impose new bindings.

Scope and Linkage

All code is defined in `*.swm` files. A `*.swm` file consists of any number of cells depending on the programmer's choice. Each file can only contain one main cell, and main cells of included files are ignored.

Project Plan

Programming Style Guide

We followed the Ten Commandments of Swarm Development:

1. Swarm is your language. You should not worship any other languages besides Swarm.
2. Do not use anything you learned from *1007: Programming in Java*. Java is not your language.
3. Do not create Cells in vain. They are powerful and should be used and created with utmost care.
4. Show up at the Sunday meetings.
5. Honor the pre-defined cells as specified. Do not create new pre-defined cells whenever you need to create a new application.
6. Respect the Swarm computational model – there's no need for fuses, breaks, or branches.
7. Do not leave code uncommented.
8. Do not try to implement algorithms from third-parties. Swarm is your language and it breaks all norms.
9. Do not use unnecessary whitespace.
10. Do not commit non-compiling code.

Because Swarm is radically different from most languages, both in syntax and in semantics, the most significant difficulty [and hence most binding programmer convention] is to conform to the unique style of Swarm. There were a few times where an object-oriented trick may have worked, but those tricks go against the essential values of the language.

Commenting is essential, but even more importantly code should be readable. Descriptive variable, object, facet, and cell names ensure that no one will be confused by the results. Comments ensure that we are all working in the same direction.

Consistent coding style is essential, so that diff's are meaningful and the code appears the same in all editors. We decided to standardize on Eclipse to avert any confusion on style, using vanilla editor settings. It also provided us with a convenient environment for debugging.

Project Timeline

September 12, 2007	Hammer out initial idea and assign team roles.
September 25, 2007	Create and submit a project proposal.
October 10, 2007	Have Subversion working as well as a lexer for out language.
October 18, 2007	Complete the parser and the LRM.
November 15, 2007	Have basic "Hello World" working. Create killer apps to work toward.
December 1, 2007	Implement first killer app. Begin debugging and testing.
December 15, 2007	Code freeze. Complete final proposal and prepare for presentation.

Software Development Environment

Our group made use of the publicly available Subversion repositories on Google Code. We used Google Docs to simultaneously edit the Language Reference Manual. Our IDE of choice was Eclipse with ANTLR and JAVA support. The ANTLR plugin for Eclipse allowed us to generate JAVA code within the IDE. We used Eclipse's debugging functionality for testing.

Roles and Responsibilities

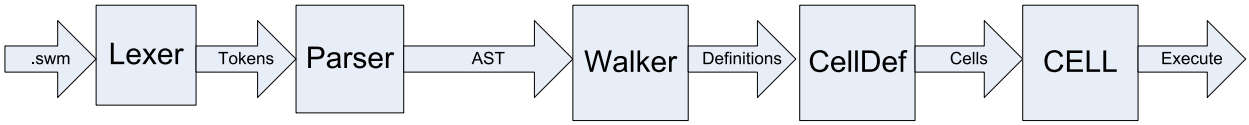
Gregory Bramble	Greg worked mainly in documentation and debugging. He created the Final Project Report as well as lending some assistance in the backend of Swarm.
Thomas Chau	Swarm is Thomas' brainchild. He created the vision and was instrumental in defining the syntax and the front-end. He kept the team motivated with interesting information about outside work in cellular computer science.
Jason Gluckman	Jason worked on Swarm's backend. He worked on our killer app, Conway's Game of Life, in Watt 2K with Thomas almost all of finals study week.
Rajesh Ramakrishnan	Rajesh was team leader. He organized meetings and created the Language Resource Manual. He wrote the lexer and parser and designed many of the algorithms on a conceptual level.

Project Log

September 12, 2007	First team meeting. Discussed Thomas' vision. Roles were given and the initial design process began.
September 19, 2007	Renamed the language. Began work on the project proposal.
September 25, 2007	Proposal submitted.
October 1, 2007	Feedback received from Phong. Work began on LRM.
October 10, 2007	Subversion server was set up. The lexer was completed. Work began on Parser.
October 16, 2007	The parser was completed. Our test program now generated an abstract syntax tree.
October 18, 2007	Swarm's Language Reference Manual was completed.
November 1, 2007	Feedback received from Phong. Work began on intermediate representation generated by a tree walker.
November 30, 2007	"Hello World" worked!
December 9, 2007	The IR was created for the Conway cell.
December 11, 2007	Built-in cells worked. We added numbers.
December 13, 2007	Individual Conway cells ran properly. A DOT representation of the Swarm program was properly generated.
December 17, 2007	Final Report was completed.

Architecture Design

The Swarm implementation is a pure interpreter. The overall flow is shown below:



The programmer writes the .swm file with the code. The Lexer [programmed in ANTLR] takes the code and produces tokens that the Parser [programmed in ANTLR] uses to produce an abstract syntax tree. The walker reads the tree and builds cell definitions. Once those are constructed, the Cell Definitions are actually realized and the cell is executed.

The `main` method controls the transition between the overarching steps. ANTLR-produced java classes run the first three steps. A cascade of java classes handles the rest of the steps.

The last two stages are interesting: here, the IR structure is invoked to build up a network of functional units, and ensuring that the structure is exactly as specified. Once the lattice is built and all cells have been constructed as well as linked up, the computation is ready to begin. Note a key difference between Swarm and most other languages: *the network is the computation.* Rather than executing expressions and statements as direct results, Swarm programs generate the structure of cells, which then produce the desired computation. In a sense, we utilize a '*cellular virtual machine*' that emulates our computational model on top of the typical Turing-Machine-like computers we use today. This platform frees us to use such higher level abstractions.

The actual execution mechanism works by propagating data through a distributed network of binds connecting facets of cells. Values are propagated from facet to facet, with the receiver being '*satisfied'* when it has received a value from every neighbor defined. Then the facet internalizes these values, passing them to the cell inside to handle. When the cell processes the data, it goes through the same process of managing data flow and *exports values from all output facets when the entire unit's computation is finished.* By waiting upon every encapsulated unit to finish every branch inside itself before allowing it to release, an application programmer may observe much more predictable behavior; the variance of lengths of branches will not influence how a programmer must organize a cell.

Note also that *facets represent data dependencies*: a facet must receive *all* values from its neighbors before it can trigger its owner cell to accept the values. Such behavior is analogous to waiting between threads for combining dependent data. A related subtlety to also note is that *facets need not wait on each other* – each facet stands for a different thread and may be stimulated independently. This frees the programmer from considering dependencies that may be *inside* a cell.

When a cell is stimulated by one of its input facets, all other input facets are ‘nillified’ – that is, they have ‘nil’ values placed on them so that when they export data, it will be ignored by the subcells. The cell itself manages dataflow, synchronizing events and ensuring that the data flows through the structure until it impinges upon an outbound facet. When all outbound facets are satisfied, the cell finally then releases their values to the exterior.

To implement new core cells, one would extend `run.Cell`, overriding:

- getFacets: returns a list of all of the cell’s facets
- stimulate: called when a facet is ready, the cell can now execute.
- [optional] toDot: returns a DOT representation of the cell (recursively called to produce pictures)

Testing Plan

Simple Test Cases

For each cell, ensure that there are fully-developed test cases. Fully-developed test case means that every detail, from the tokens to the AST to the IR to the execution, is known in advance. Any sort of non-determinism will come back to haunt later on. For example, the three cells below are part of the test suite for the INT cell.

```
[INT:"5"->SOUT; INT:"2"->SOUT]:MAIN;

[(INT:"5", INT:"2", INT:"-10")->ADD->SOUT]:MAIN;

[(.in)->ADD:s->(.out); INT:"3"->s;]:ADDTHREE;
[INT:"2"->(.in)(ADDTHREE)(.out)->SOUT]:MAIN;
```

The first case demonstrates that the cell is aware of its value. The second demonstrates that the INT cell has an actual meaning. The third demonstrates that values are correctly passed through facets of user-defined cells.

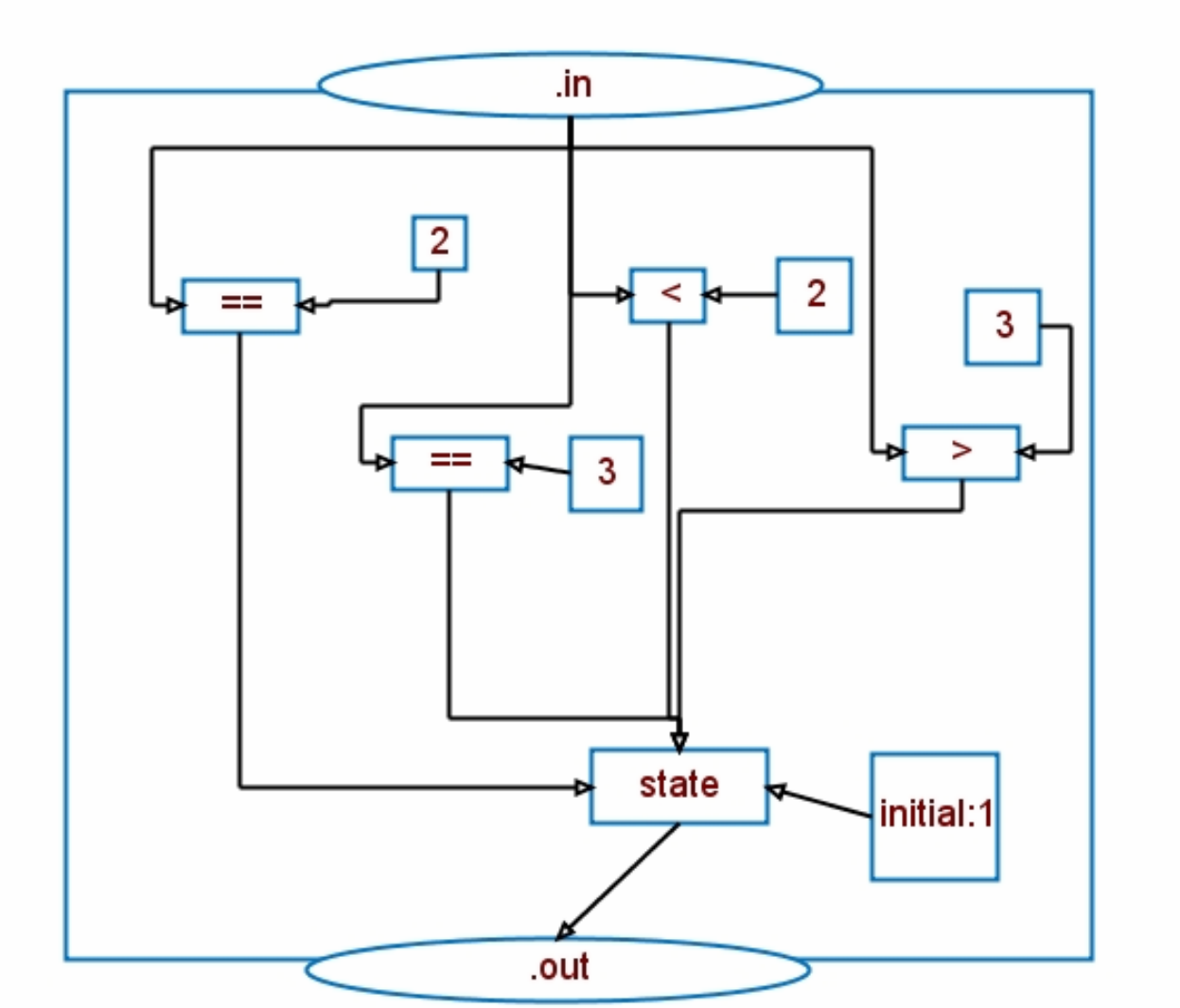
We had one target cell in mind, namely the Conway game-of-life cell. This cell, in a way, represented everything we needed:

```
[
/* Trigger to start things off */
(.clk)->STR:"";

INITIAL:"1"->.init(MEM:m);

(.in)-> ADD ->
       (.left(EQUALS:eqtwo),
       .left(EQUALS:eqthree),
       .left(LESS_THAN:lttwo),
       .left(GREATER_THAN:gtthree));
INT:"2"->(.right(eqtwo),
               .right(lttwo));
INT:"3"->(.right(eqthree),
               .right(gtthree));
(eqthree).output -> .set(m);
(lttwo).output -> NEGATE -> .set(m);
(gtthree).output -> NEGATE -> .set(m);
(eqtwo).output-> .release(m);

(m).val->(.out);

]:CONWAYALIVE;

[
/* Trigger to start things off */
(.clk)->STR:"";

/* If 2 neighbors alive, continue with this state (transmit contents of memory.
       If 3 neighbors alive, come to life (if dead). Stay alive otherwise.
       If fewer than 2 neighbors alive, die.
       If more than 3 alive, die.
 */
(.in)->ADD->(.left(EQUALS:eqtwo), .left(EQUALS:eqthree), .left(LESS_THAN:lttwo),
.left(GREATER_THAN:gtthree));
INT:"2"->(.right(eqtwo), .right(lttwo));
INT:"3"->(.right(eqthree), .right(gtthree));
(eqthree).output -> .set(MEM:m);
(lttwo).output -> NEGATE -> .set(m);
```

```
(gtthree).output -> NEGATE -> .set(m);
(eqtwo).output-> .release(m);

/* Initialize state to DEAD */
INITIAL:"0"->.init(m);

/* Output the state */
(m).val->(.out);

]:CONWAYDEAD;


/* MAIN driver Cell:
Instantiate Conways (DEAD or ALIVE) and put them in a 2-Dimensional mesh.
Ensure that four sides (and DIAGONALS TOO!!!) are connected.
E.g.

*/
[
(CONWAYDEAD:a).out->(.in(CONWAYALIVE:b), .in(CONWAYALIVE:c), .in(CONWAYALIVE:d));
(b).out->(.in(a), .in(c), .in(d));
(c).out->(.in(a), .in(b), .in(d));
(d).out->(.in(a), .in(b), .in(c));

INT:"1"->(.clk(a), .clk(b), .clk(c), .clk(d));

/*
Concatenate coordinates to the state (CONWAY*).out gives 1/0 as alive/dead state
be warned: it doesn't come out in a particular order consistently (e.g. the print
statements aren't in sequence)
*/
(a).out->STR:"1,1:";
(b).out->STR:"1,2:";
(c).out->STR:"2,1:";
(d).out->STR:"2,2:"
]:MAIN;
```

When a file was parsed, every cell definition was read. This allowed us to test multiple cells at once.

Conway's Game of Life, fully implemented



The "killer app" in our project test suite is Conway's Game of Life. This application shows off many unique aspects of Swarm. The code definitions for the Conway cells are generated using generate.pl which is a script we wrote to quickly generate new cells while not abandoning the dataflow model in Swarm. Therefore, these Swarm cells may seem unreasonably large. While surely in Swarm's later releases there will be a more concise way to specify an interesting Conway cell, we felt it important to keep it clear in the code that cells are propagating data to each other through facets. Therefore, we used the Perl script to generate lots of Swarm code, and it works very well.

The full implementation of Conway also includes use of the GUI cell. This is a built-in cell which has static state. Once it is created, it maintains its elements until all of its elements have been accessed, at which point it clears itself for repopulation with data. While tailored for this Game of Life application, GUI will work successfully in any application that requires a 15x15 panel for either still or animated pictures.

Swarm files have been created for three unique states in Conway. These files are exploder.swm, glider.swm, and tumbler.swm. It is easy to create your own Conway cells using the generate.pl script. Follow these directions to do so:

1. In generate.pl, edit the 15x15 matrix. DO NOT change its dimensions.
   But you can edit the 0's and 1's all you want. This way, establish the initial Conway starting state.

2. Then, run the script. The output is in out.dat.

3. Copy the contents of out.dat into glider.swm, overwriting the main cell. Then save it as whatever name you wish.

4. That's it! Go ahead and run TestGrammar, passing the new .swm file.

Lessons Learned

Gregory Bramble

“Don’t take many other classes, especially if you are going to be really ambitious with your language design. Make sure you understand what ANTLR is going to give you and keep yourself on the same page as the rest of your team members. Working on a team is not easy.”

Thomas Chau

“Work it all out in thought before writing a single line of code. Layers of bolted-on hacks will only kill you. Get real cozy with the debugger. Plan everything in advance, in detail. Produce examples (and FULL examples) and walk through them before coding.”

Jason Gluckman

“How inherently linear computers are - it was difficult to implement a language that was parallel in nature on the fundamentally serial computers of today. Also, in the process of debugging the execution process, it was interesting to see the data spreading across the cells like an infection. Maybe a more refined version could do medical or economic simulations...”

Rajesh Ramakrishnan

“I wish someone told me ahead of time how much work was actually required to get the project rolling. Taking 9 classes on top of PLT this semester was not a good idea. Plan on everything breaking, plan on everything falling apart at least twice before the end. Expect many sleepless nights. Eschew the temptations to succumb, and you will succeed. But success requires much more time than you would think.”

Appendix: Grammar

The following grammar is derived from our .g files:

```
END: ';';
NEW: ':';
LPAREN: '(';
RPAREN: ')';
LBRACKET: '[';
RBRACKET: ']';
COMMA: ',';

CROSSBIND: "->";
DIRECTBIND: "=>";
KILL: "-|";
PUSH: ">>";
BINDTYPE: CROSSBIND | DIRECTBIND;

DIGIT: ('0'..'9');
UPPER: ('A'..'Z');
LOWER: ('a'..'z');
DIGIT_HEX : (DIGIT|'a'..'f'|'A'..'F') ;

OCTAL: ('0'..'3') (DIGIT (DIGIT)? )? | ('4'..'7') (DIGIT)?;
UNICODE : 'u' DIGIT_HEX DIGIT_HEX DIGIT_HEX DIGIT_HEX;

WS: ( ' ' | '\t' | "\r\n" | '\r' | '\n');
ESCAPE :   '\\' (('b'|'t'|'n'|'f'|'r'|'"' /* " */ |'\''|'\\')
       | OCTAL | UNICODE);

STRING :  '"'! ( ESCAPE | ~('\\'|'"') )* '"'! ;

COMMENT: "/*"
         ( options { generateAmbigWarnings=false; }
           : { LA(2)!='/' }? '*'
           | '\r' '\n' {newline();}
           | '\r' {newline();}
           | '\n' {newline();}
           | ~('*'|'\n'|'\r')
         )*
         "*/" {$setType(Token.SKIP);}
         ;

CELLTYPE: (UPPER) ((UPPER) | '_' )* ;
INSTANCE: (LOWER)+ | (DIGIT)+ ;
FACET: '.' (LOWER) ((LOWER) | '_' | (DIGIT))* ;
```

```
/* Plain old Facet */
facetunit  : FACET;

/* Facet with post - (identification | definition)
 * Added to resolve .a -> (.left(EQUALS:eq), .b) */
facetdef : facetunit (LPAREN! unit RPAREN! (facetunit)?)?;

/* Isolated or groups of facets */
facetblock : facetunit
           | LPAREN! facetdef (COMMA! facetdef)* RPAREN!;

/* Cell-level specifiers */
protected
unit : CELLTYPE (NEW! INSTANCE
         | NEW! STRING)
     | anon_def;

/* Statement-level blocks */
block: facetblock (LPAREN! unit RPAREN! (facetblock)?) | unit
     | LPAREN! unit ( (RPAREN! (facetblock)? )
                    | ((COMMA! unit)+ RPAREN!));

/* Statements */
statement: block (bindtype block)* ;

/* Anonymous Cells */
anon_def: LBRACK! statement (END! statement)* (END!)? RBRACK!

/* Cell Definitions */
definition: anon_def NEW! celltype END!
```

Appendix B: Source Code

Repository Details

The source repository can be accessed at http://code.google.com/p/swarm-language/
The following code snapshot is the frozen version before the final presentation.

Code Listing

The Lexer, Parser, Walker, and Main

SwarmLexer.g

```
/* vim: set tabstop=3: */
/**************************************************************************
 * SwarmLexer.g : Lexer and Stub Parser for the Swarm Language
 *
 * SAMPLE USAGE:
 * public class TestGrammar {
 *    public static void main(String[] args) throws Exception {
 *       SwarmLexer lex = new SwarmLexer(System.in);
 *       SwarmParser parse = new SwarmParser(lex);
 *       parse.expr();
 *       System.out.println("Success!");
 *    }
 * }
 *
 * Consult the LRM for more information.
 * @author Rajesh              */

class SwarmLexer extends Lexer;

options {
       k = 2;
       exportVocab=Swm;
}

CELLTYPE: (UPPER) ((UPPER) | '_' )* ;

INSTANCE: (LOWER)+ (DIGIT)*
        ;

FACET: '.' (LOWER) ((LOWER) | '_' | (DIGIT))* ;

protected
CROSSBIND: "->";

protected
DIRECTBIND: "=>";

END: ';';

NEW: ':';

LPAREN: '(';

RPAREN: ')';

WS: ( ' ' | '\t'
    |  ("\r\n" | '\r' | '\n'){ newline(); }
    ) { $setType(Token.SKIP); };
```

```
protected
ESCAPE:   '\\' (('b'|'t'|'n'|'f'|'r'|'"' /* " */ |'\''|'\\') | ESCAPE_OCTAL | ESCAPE_UNICODE);

protected
ESCAPE_OCTAL :  ('0'..'3') (options{warnWhenFollowAmbig=false;}
                : DIGIT (options{warnWhenFollowAmbig=false;}: DIGIT)? )?
             | ('4'..'7') (options{warnWhenFollowAmbig=false;}: DIGIT)?;

protected
ESCAPE_UNICODE : 'u' DIGIT_HEX DIGIT_HEX DIGIT_HEX DIGIT_HEX;

COMMA: ',';

LBRACKET: '[';

RBRACKET: ']';

protected
KILL: "-|";

protected
PUSH: ">>";

protected
DIGIT: ('0'..'9');

protected
DIGIT_HEX : (DIGIT|'a'..'f'|'A'..'F') ;

protected
UPPER: ('A'..'Z');

protected
LOWER: ('a'..'z');

BINDTYPE: CROSSBIND | DIRECTBIND /* | KILL | PUSH */;

COMMENT: "/*"
         ( options { generateAmbigWarnings=false; }
           : { LA(2)!='/' }? '*'
           | '\r' '\n' {newline();}
           | '\r' {newline();}
           | '\n' {newline();}
           | ~('*'|'\n'|'\r')
         )*
         "*/" {$setType(Token.SKIP);}
         ;

STRING :  '"'! ( ESCAPE | ~('\\'|'"') )* '"'! ;
```

SwarmParser.g

```
/***************************************************************************
       Swarm Parser
       @author Rajesh
***************************************************************************/
class SwarmParser extends Parser;
options {
       k = 2; /* no ambiguity with bind types */
       buildAST=true; /* we want the AST! */
       importVocab=Swm;
}
/* literalize these: */
tokens {
```

```
        DEFINITION;
        STATEMENT;
        BLOCK; /* Treat all blocks equally */
        BIND;
}

/* Plain old Facet */
protected
facetunit  : FACET;

/* Facet with post - (identification | definition)
 * rr2318: added to resolve .a -> (.left(EQUALS:eq), .b) */
protected
// need separators between units...d
facetdef : facetunit (LPAREN! unit {#facetdef = #([BLOCK,"BLOCK.UNIT_FACET"],#facetdef);} RPAREN!
(facetunit)?)?;

/* Isolated or groups of facets */
protected
// BLOCKS FOR FACETS??!?!
facetblock : facetunit {#facetblock = #([BLOCK,"BLOCK.ONE_FACET"],#facetblock);}
           | LPAREN! facetdef (COMMA! facetdef)* RPAREN!
           {#facetblock = #([BLOCK,"BLOCK.FACETS"],#facetblock);};

/* Hack to add to AST */
protected
bindtype : BINDTYPE
           {#bindtype = #([BIND,"BIND"],#bindtype);};

/* Hack to add to AST */
protected
celltype: CELLTYPE {#celltype = #([BLOCK,"BLOCK.UNIT.ASSIGNMENT"],#celltype);};

/* Cell-level specifiers */
protected
unit : CELLTYPE (NEW INSTANCE {#unit = #([BLOCK,"UNIT.LABEL"],#unit);}
         | NEW! STRING {#unit = #([BLOCK,"UNIT.STRING"],#unit);}
         | /* nothing */ {#unit = #([BLOCK,"UNIT.GENERIC"],#unit);})
     | INSTANCE {#unit = #([BLOCK,"UNIT.INSTANCE"],#unit);}
     | anon_def {#unit = #([BLOCK,"UNIT.ANON_DEF"],#unit);};

/* Statement-level blocks:
               facetblocks
               cell blocks
 */
protected
block: facetblock (LPAREN! unit RPAREN! (facetblock)?
             {#block = #([BLOCK,"BLOCK.UNIT_FACET"],#block);}
       | /* nothing */
     )
     | LPAREN! unit ( (RPAREN! (facetblock)? )
                      {#block = #([BLOCK,"BLOCK.UNIT_FACET"],#block);}
                    | ((COMMA! unit)+ RPAREN!)
                      {#block = #([BLOCK,"BLOCK.UNIT_LISTS"],#block);})
     | unit {#block = #([BLOCK,"BLOCK.UNIT"],#block);};

/* Statements */
statement: block (bindtype block)*
           {#statement = #([STATEMENT,"STATEMENT"], #statement);};

/* Anonymous Cells */
protected
anon_def: LBRACKET! statement (END! statement)* (END!)? RBRACKET!
          {#anon_def = #([DEFINITION,"DEFINITION"], #anon_def);};

/* Cell Definitions */
definition: anon_def NEW! celltype END!
    /* rr2318: uncomment next line to print out new cell types */
    /* { System.out.println("DEFINED CELL: " + left.getText());}/* IGNORE*/
```

```
          {#definition = #([DEFINITION,"DEFINITION.CELL"], #definition);};

/* For now we only check whether the input is a valid set of statements */
expr: (definition)+ EOF!;
```

SwarmWalker.g

```
{
       import ir.*;
       import ir.blocks.*;
       import java.util.*;
}
/**************************************************************************
       Swarm Tree Parser (The TreeWalker)
       tc2165: 11/2/07:
              pending:
              - finish the walk [ok]
              - actually make the walk generate an IR structure [ok, primitively sofar]
              - execute bindings [ok]
              - "hello world" [done, primitively 11/22/07]
              - general Bindables (unit, facetblock)
              - Stmt do general structures
**************************************************************************/

class SwarmWalker extends TreeParser;

options {
       importVocab=Swm;
}

/* Setup the Program: build dictionary, init MAIN cell */
{
       //HashMap<String, CellDef> dict = new HashMap<String, CellDef>();

       CellDef main = new CellDef("MAIN");
}

/* Files: a sequence of cell declarations (must include a MAIN cell) */
file returns [CellDef main] {
              CellDef a;
              main = null;
       // add to a data struct ArrayList<Named> for Named interface (implements named) to manage
the cells table.
       // main cell
              //System.out.println("File");
        }
       : (a=cell {
              // if cell is main, set as main. add to dict.
              if (#a.getName().equals("MAIN")) {
                     main = #a;
              }
              // else just add to dict.
              CellLib.getInstance().addCellDef(a);

       })+;

/* Cells: a definition and name */
cell returns [CellDef r]
              {
                     r = null;
                     String cell_type;
                     StructuralDefinition d;
              }
       : #(DEFINITION d=definition cell_type=unit_assignment ) {
              CellDef def = new CellDef(#cell_type);
              def.setStructure(#d);
              r = def;
```

```
        };

/* Structural Definition (Lattice): sequence of statements */
definition returns [StructuralDefinition d] {
                        Stmt s;
                        d = new StructuralDefinition();
                        //System.out.println("Construct structural definition.");
                }
        : #(DEFINITION (s=statement {
                                                        /* Add every statement to the definition */
                                                        d.add(s);
                                                }
                                        )+);

/* Statements within a cell definition: a chain of binds */
statement returns [Stmt r] {
                        r = null;
                        BindType c;
                        Block b, d;

                        java.util.ArrayList<Object> list = new java.util.ArrayList<Object>();
                }

                :#(STATEMENT
                        (b=block {
                                //System.out.println("STMT    Block: " + #b);
                                        list.add(#b);
                                        } |
                         c=bind {
                                        //System.out.println("STMT    Bind: " + #c);
                                        list.add(#c);
                                }
                         )+
                        ) {
                                /* Give the chain to a Stmt to analyze */
                                r = new Stmt(list);
                        };

/* Different types of blocks
BLOCK.UNIT, BLOCK.FACETS , BLOCK.UNIT_LISTS, BLOCK.ONE_FACET ,  */
block returns [Block r] {
                        FacetsBlk fleft, fright;
                        CellUnit u;
                        Block b = r = null;
                        r = new UnitListsBlk();
                }
                        /* BLOCK.UNIT */
                        /* or UNIT_LISTS: list of bindables */
        : #(BLOCK {
                        //System.out.println("Block Type: " + #BLOCK.getText());
                        r.setType(#BLOCK.getText());
                }
                (

                   /* Blocks are just lists of Blocks */
                  (b=block {
                                /* add sub-block to block */
                                r.add(#b);
                        })+             |

                  /* ********************** */
                  /* In the terminal cases:  */
                  /* ********************** */
                  /* Boils down to:  */

                  // UnitBlk, UnitFacetBlk, FacetsBlk, or UnitListsBlk
            ((u=unit {
                                // It's a UnitBlk
                                //System.out.println("   adding sub unit to block");
```

```
                                        //System.out.println("UnitBlk");
                                        r = new UnitBlk();
                                        r.add(#u);
                        } ((fright=facetblock {
                                                //System.out.println("UnitFacetBlk");
                                                // Actually, it's a UnitFacetBlk
                                                UnitBlk tmp = (UnitBlk) r;
                                                r = new UnitFacetBlk();
                                                ((UnitFacetBlk) r).setUnit(tmp);
                                })? |
                                // No, wait, it's REALLY a unit list blk.
                                (u = unit {
                                                //System.out.println("UnitListsBlk");
                                                r = new UnitListsBlk();
                                                r.add(#u);
                                 })+)
                              )) |

                /* UNIT_FACET: (.a,b)BLAH(.c,.d) */
                /* TODO: associate the facetblocks with the unit CONTEXTUALIZING them. return
bindable block. */
                (fleft=facetblock {
                                                // a simple facetblock
                                                r.add(#fleft);
                                        }
                                        (b=block {
                                                        // or may be a unitfacet block
                                                        r = new UnitFacetBlk((UnitBlk) #b);

                                                        ((UnitFacetBlk) r).setLeft(#fleft); //
preserve the left side
                                                }
                                                (fright=facetblock {
                                                        // add right facetblock to the unitfacet
block
                                                        ((UnitFacetBlk) r).setRight(#fright);
                                                })?
                                        )?) //(COMMA b=block {r.add(#b);})+
                           )) {

                                // in case of trivial blocks, flatten the structure
                                Block possible = null;
                                possible = r.getFirstSubBlock();
                                if (possible != null) {
                                        r = possible;
                                }
                           };

/*      Units are: (CELLTYPE:instance}
                   instancename
*/
unit returns [CellUnit r] {
                        r = null;
                }
        :  (CELLTYPE {
                                /* "static" cells */
                                //System.out.print("unit: CELLTYPE:" + #CELLTYPE.getText() + " || "
);

                                // RETURN A BINDABLE CELL_DEF. (later, CELL_DEF will gen() to
produce a Cell obj)

                                //r = new run.core.Sout();
                                r = new CellUnit(#CELLTYPE.getText());
                        }
                        /* Optionally, named/instanced */
                        (NEW INSTANCE {
                                //System.out.println( "   [instance: " + #INSTANCE.getText() +
"]");
```

```
                        r.setLabelName(#INSTANCE.getText());
                }|
                (STRING {
                        //System.out.println(" unit: String : " + #STRING.getText());
                        r.setLabelName(#STRING.getText());

                        }))? |
        /* Or, instances of cells declared prior */
        (INSTANCE {
                        // TODO: an instance resolution
                        r = new InstanceRef(#INSTANCE.getText());

                        //System.out.println("unit: Named instance : " +
#INSTANCE.getText());
                }));

/* TODO: a singular facet unit, or a list of facetdefs e.g. ( .a(CELL).b, .c(CELL2).d )  */
facetblock returns [FacetsBlk b] {
                b = null;
                java.util.ArrayList<String> facetList = new java.util.ArrayList<String>();
        }
        : (FACET {
                /* Add facet to list of facets */
                facetList.add(#FACET.getText());
        })+
                {
                        b = new FacetsBlk(facetList);
                };

bind returns [BindType r] {
        r=null;
        //System.out.print("\n           Bind:");
}
        : #(BIND r=bindOp);
bindOp returns [BindType r] { r = null; }
        : BINDTYPE {
                //System.out.println(#BINDTYPE.getText() + "\n");
                r = new BindType(#BINDTYPE.getText()); /* return new bind based on the text: ->
or => */
                };

/* Assigning definitions to units */
/* Relate the lattice to a name (TODO) */
unit_assignment returns [String name] {
                name = null;
        }
        : #(BLOCK CELLTYPE) {
                //System.out.println("Structural Definition associated with Unit Assign:" +
t.toStringTree());
                name = #CELLTYPE.getText();
        };
```

TestGrammar.java

```
import ir.CellDef;

import java.io.File;
import java.io.FileInputStream;

import run.Cell;

import antlr.DumpASTVisitor;
import antlr.collections.AST;
public class TestGrammar {
```

```
	private static final boolean STDIN_USING = false;

	public static void main(String[] args) throws Exception {
		SwarmLexer lexer;

		/* Lex */
		if (STDIN_USING)
			 lexer = new SwarmLexer(System.in);
		else {
			FileInputStream swarmProgram = new FileInputStream(new File(args[0]));
			lexer = new SwarmLexer(swarmProgram);
		}

		/* Parse */
		SwarmParser parser = new SwarmParser(lexer);
		parser.expr();

		System.out.println("Parsed");
		/* Walk AST */
		AST ast = (AST)parser.getAST();
		DumpASTVisitor visitor = new DumpASTVisitor();
		visitor.visit(ast);
		SwarmWalker walker = new SwarmWalker();

		CellDef mainCellDef = walker.file(ast);

		Cell realizedMain = mainCellDef.realize();
		realizedMain.exec();
		realizedMain.toString();
	}
}
```

Binds, Facets, Cells

Bind.java

```
package run;

import ir.BindType;

public class Bind {

	private Facet left;
	private BindType bindType;
	private Facet right;

	public Bind(Facet left, BindType bindType, Facet right) {
		this.left = left;
		this.bindType = bindType;
		this.right = right;

		left.addDestination(right);
		right.addSource(left);
	}

	public Facet getLeft() {
		return left;
	}

	public Facet getRight() {
		return right;
	}

	public BindType getBindType() {
		return bindType;
	}
```

```
	public String toString() {
		return left.toString() + " " + bindType.toString() + " " + right.toString();
	}

}
```

BindTable.java

```
package run;
import java.util.AbstractMap;
import java.util.ArrayList;
import java.util.ListIterator;

import run.client.Bindable;

public class BindTable {
	ArrayList<TailBind> tb;
	ArrayList<HeadBind> hb;
	AbstractMap<String, Bindable> labelsToBindables;

	public Object step(){
		ListIterator<HeadBind> liter = hb.listIterator();
		while(liter.hasNext()){
			HeadBind next = liter.next();
			if(next.isReady()){
				//isReady will check if all of it's tailbinds are done
				//then call act on Bindable
			}
		}
		return null;
	}
}
```

Cell.java

```
package run;

import java.util.ArrayList;
import java.util.HashMap;
import java.util.HashSet;
import java.util.LinkedList;
import java.util.List;
import java.util.Queue;
import java.util.Set;

import run.core.NIL;

/**
 * Abstract class which represents the generalized Cell. All other functional cells must
 * extend this class. The Cell is responsible for managing bindings, facets, and
 * intracellular transportation of data subcells.
 *
 * User-defined cells "speciate" from this stem cell by providing their own
 * Structural Definition (a blueprint) that is realized into an actual Lattice of
 * subcells that govern the cell's behavior.
 *
 * All Cells perform a computation when stimulated on their (satisfied) facets.
 *
 * Cells manage the set of facets that they own and manage the flow of computation
 * through their internal subcell structure -- the lattice.
 *
 *
 * @author Thomas Chau
 *
 */
public class Cell  {

	protected boolean INPUT_FACET = true;
	protected boolean OUTPUT_FACET = false;


	protected String cellTypeName;
	protected String labelName;
	private Lattice lattice;

	/* Facets ready to export */
	private Queue<Facet> outReadyFacets = new LinkedList<Facet>();

	//Hashmap of all this cell's facets from name -> facet
	protected HashMap<String, Facet> facets = new HashMap<String, Facet>();

	//ArrayList of all this cell's output facets
	private ArrayList<Facet> outFacets = new ArrayList<Facet>();
	//ArrayList of all this cell's input facets
	private ArrayList<Facet> inFacets = new ArrayList<Facet>();

	// debug flag
	private boolean reporting = false;


	public Cell(String typeName, String labelName) {
		this.cellTypeName = typeName;
		this.labelName = labelName;
	}

	public Cell(String name, Lattice lattice) {
		super();
		this.cellTypeName = name;
		this.lattice = lattice;
	}

	public void realize() {

	}
```

```
	public void exec() {
		report("\n ********************** \n Writing DOT " + cellTypeName);
		DotConverter dot = new DotConverter("swarm.graph");
		dot.startDotFile();
		dot.convert(this);
		dot.endDotFile();

		report("\n ********************** \n Executing " + cellTypeName);
		stimulate(null);
	}

	/**
	 * Special Case: for MAIN, stimmer will be NULL
	 * STIMMER IS A -BORDER- facet of THIS cell.
	 * @returns the final output facets that this stimulation deposited onto,
	 * awaiting export to neighbors
	 */
	public List<Facet> stimulate(Facet stimmer)
	{
		Set<Facet> ourOutFacetsRdy = new HashSet<Facet>();

		// MAKE ALL OTHER INFACETS OF THE CELL GIVE NIL TO THEIR BRANCHES
		for (Facet altInFacet : inFacets) {
			if (altInFacet != stimmer) {
				altInFacet.nillify();
				outReadyFacets.add(altInFacet);
			}
		}

		// Intrinsic Cells need to push their OutFacets to the OUTREADY queue --
		// e.g. dangling ints, initializers, etc.
		Set<Cell> subcells = lattice.getSubcells();
		for (Cell intrinsic : subcells) {
			if (intrinsic.isLiteral())
				outReadyFacets.addAll(intrinsic.getOutFacets());
		}

		outReadyFacets.add(stimmer);

		// While OUTREADY facet queue (facets with outbound ready data) isn't empty,
		while (outReadyFacets.size() > 0) {
			// pop a facet off the OUTREADY
			Facet nextRdy = outReadyFacets.remove();
			while (nextRdy == null && !outReadyFacets.isEmpty())
				nextRdy = outReadyFacets.remove();
			if (outReadyFacets.isEmpty() && nextRdy == null)
				break;

				//	propagate values to where they belong (according to Lattice
def)
				//		(stimulations will occur between here and
				//	facets will add themselves to end of a 'ready' list in the
parent cell)
				// - done output facets of a subcell that just finished (add to
outReadyFacets)
				// - distinguish these from border facets of THIS going out.

			// port the values from the stimulating facet
			List<Object> outboundValues = nextRdy.dispatchOutbound();

			// bring them to the destination facets, replicating the data for each
			List<Facet> destinations = lattice.getDestinations(nextRdy);

			// if any of the destinations belong to us, DONE -- it's one of the facets
			// we should give to enclosing cell as outReadyFacet.
			for (Facet dest : destinations) {
				boolean facetReady = dest.receive(nextRdy, outboundValues);

				// if it's one of our exiting border facets
				if (dest.getOwner() == this) {
					ourOutFacetsRdy.add(dest);
```

```
                                }
                                else if (facetReady) {
                                        // stimulate, add subcell's ready output facets to be
propagated next.
                                        List<Facet> subcell_doneFacets =
dest.getOwner().stimulate(dest);
                                        for (Facet f : subcell_doneFacets) {
                                                outReadyFacets.add(f);
                                        }
                                }
                        }
                }

                List<Facet> outs = getOutFacets();
                for (Facet f : outs) {
                        if (f.isReady()) {
                                ourOutFacetsRdy.add(f);
                        }
                }

                ourOutFacetsRdy = stripNils(ourOutFacetsRdy);
                return new LinkedList<Facet>(ourOutFacetsRdy);
        }

        private Set<Facet> stripNils(Set<Facet> ourOutFacetsRdy) {
                for (Facet f : ourOutFacetsRdy) {
                        f.clearNils();
                }
                return ourOutFacetsRdy;
        }

        public List<Facet> getFacets() {
                return new ArrayList<Facet>(facets.values());
        }

        /**
         * List of input facets of this cell.
         * @return
         */
        public List<Facet> getInFacets() {
                return inFacets;
        }

        /**
         * List of output facets of this cell.
         * @return
         */
        public List<Facet> getOutFacets() {
                return outFacets;
        }

        public void addFacet(Facet f, boolean isInputFacet)
        {
                report("   Adding facet " + f + " to " + toString());
                facets.put(f.getName(), f);
                if (isInputFacet)
                        inFacets.add(f);
                else
                        outFacets.add(f);
        }


        //Prints out the dot syntax for this cell
        public String toDot(String prefix)
        {
                String retString = "";
                /*retString += "subgraph cluster_" +  prefix + cellTypeName + hashCode() + " {\n";

                retString += "label = \"" + cellTypeName + "\";\n";
                HashSet<Cell> subcells = new HashSet<Cell>();
```

```
		for(Facet f : dependencies.keySet())
		{
			subcells.add(f.getOwner());
		}

		subcells.remove(this);

		//Make all the subgraphs
		prefix += cellTypeName + hashCode();
		for(Cell c : subcells)
		{
//			retString += c.toDot(prefix);
		}

		//Print my facets with proper, simple labels
		for(Facet f : facets.values())
		{
			retString += prefix + f.getName().substring(1) + " [label = \"" +
f.getName() + "\"]\n;";
		}

		//Put in all the bindings of this cell
		if (lattice != null) {
			for(Bind b : lattice.getBindings())
			{
				Facet left = b.getLeft();
				Facet right = b.getRight();

				boolean toggleLeft = false;
				boolean toggleRight = false;
				if(left.getOwner() == this)
					toggleLeft = true;
				if(right.getOwner() == this)
					toggleRight = true;

				retString += prefix;
				if(!toggleLeft)
					retString += left.getOwner().cellTypeName +
left.getOwner().hashCode();
				retString += left.getName().substring(1);
				//retString += " [label = \"" + left.getName().substring(1) + "\"]
";
				retString += " -> ";
				retString += prefix;
				if(!toggleRight)
					retString += right.getOwner().cellTypeName +
right.getOwner().hashCode();
				retString += right.getName().substring(1);
				//retString += " [label = \"" + right.getName().substring(1) + "\"]
";
				retString += ";\n";
			}
		}
		retString += "}\n";*/
		return retString;
	}

	public void report(String s) {
		if (reporting)
			System.out.println(s);
	}

	public boolean isLiteral()
	{
		return false;
	}

	public String toString() {
		return cellTypeName;
	}
```

```
	public void setLabelName(String labelName) {
		this.labelName = labelName;
	}

}
```

DotConverter.java

```
package run;
import java.io.BufferedWriter;
import java.io.IOException;
import java.io.FileWriter;

//Converts a cell to a dot file
public class DotConverter {

	BufferedWriter writer;
	public DotConverter(String filename)
	{
		try
		{
			writer = new BufferedWriter(new FileWriter(filename));
		}
		catch(IOException e)
		{
			System.out.println("ERROR!!!! CAN'T WRITE DOT GRAPH!!!!");
		}
	}

	public void startDotFile()
	{
		try
		{
			writer.write("digraph G {\n");
		}
		catch(IOException e)
		{
			System.out.println("ERROR WHILE WRITING DOT GRAPH");
		}
	}

	public void endDotFile()
	{
		try
		{
			writer.write("}\n");
			writer.close();
		}
		catch(IOException e)
		{
			System.out.println("ERROR WHILE WRITING DOT GRAPH");
		}
	}

	public void convert(Cell c)
	{
		try
		{
			String prefix = "x";
			writer.write(c.toDot(prefix));
		}
		catch(IOException e)
		{
			System.out.println("ERROR WHILE WRITING DOT GRAPH");
		}
	}
}
```

Facet.java

```
package run;

import java.util.ArrayList;
import java.util.HashMap;
import java.util.LinkedList;
import java.util.List;
import java.util.Map;
import java.util.Queue;

import run.core.NIL;


/**
 *
 * Facets are contextualized to an owner cell and have a handle
 * on both the enclosing (client/user) cell and their owner.
 * Facets manage the flow of data, acting as gatekeepers on borders,
 * only permitting penetration of cellular membranes when
 * the data conditions are met.
 *
 * This is called "Bind and Wait".
 * On the INBOUND direction:
 *             If a cell facet is used by an enclosing cell that creates
 *             five different binds on it, the facet will wait until it has
 *     received values from ALL FIVE SOURCES before allowing
 *     absorption of those values into its owner.
 *
 *  On the OUTBOUND direction:
 *      Similar restrictions apply against values coming from the
 *      interior of the cell, with an additional condition:
 *             Values will only be released when the current computation
 *             has completed every branch.
 *
 *
 *
 * ADD FACET'S HANDLE ON ITS OWNING/CONTEXTUALIZING CELL
 *
 * @author Thomas Chau
 */
public class Facet {
        private String name;
        private Cell owner;
        boolean isInputFacet;

        // places this facet might be bound to (as in, places it owes)
        List<Facet> destinations;

        // places this facet is bound to (places that owe it)
        List<Facet> sources;

        Map<Facet, Queue<Object>> inPort;

        public Facet(String name, Cell owner, boolean isInputFacet) {
                this.owner = owner;
                this.name = name;

                this.isInputFacet = isInputFacet;
                this.inPort = new HashMap<Facet, Queue<Object>>();

                destinations = new ArrayList<Facet>();
                sources = new ArrayList<Facet>();

                owner.addFacet(this, isInputFacet);
        }

        public Cell getOwner() {
                return owner;
        }
        public String getName() {
```

```
		return name;
	}

	public String toString() {
		return name + ":" + hashCode() + "(" +  owner.cellTypeName + ": " +
owner.hashCode() + ")";
	}

	/**
	 * A list of outbound data ships out. (Guaranteed ready)
	 * @return
	 */
	public List<Object> dispatchOutbound() {
		List<Object> outbound = new ArrayList<Object>();
		for (Facet f : sources) {
			Queue ourQ = inPort.get(f);

			if (ourQ.size() == 0)
				System.out.println("crap");

			Object head = ourQ.remove();
			outbound.add(head);
		}
		return outbound;
	}

	public List<Object> dispatchOutboundEntire() {
		List<Object> outbound = new ArrayList<Object>();
		for (Facet f : sources) {
			Queue ourQ = inPort.get(f);
			outbound.addAll(ourQ);
			ourQ.clear();
		}
		return outbound;
	}

	/**
	 * Accepts the values from another facet to the port.
	 *
	 * @param stimmer
	 * @param outboundValues
	 * @return whether the facet is ready after receiving the values
	 */
	public boolean receive(Facet stimmer, List<Object> inboundValues) {
		inPort.get(stimmer).addAll(inboundValues);
		boolean ready = true;

		// for all source facets
		// if they ALL have a queue size > 0 of values waiting in the dock... we're good
		for (Facet f : sources) {
			Queue<Object> inBoundFromSrc = inPort.get(f);
			if (inBoundFromSrc.size() == 0) {
				ready = false;
			}
		}

		return ready;
	}

	public void addDestination(Facet right) {
		destinations.add(right);
	}

	public void addSource(Facet left) {
		sources.add(left);
		inPort.put(left, new LinkedList<Object>());
	}

	/**
	 * Inserts NIL into all ports. (Dead Branch)
	 */
```

```
	public void nillify() {
		for (Facet f : sources) {
//			System.out.println(name + " nil " + f.name);
			inPort.get(f).add(new NIL());
		}
	}

	public void clearNils() {
		for (Facet f : sources) {
			Queue<Object> queue = inPort.get(f);
			NIL nil = new NIL();
			while (queue.contains(nil)) {
				queue.remove(nil);
			}
		}
	}

	public boolean isReady() {
		boolean ready = true;

		for (Facet f : sources) {
			Queue<Object> inBoundFromSrc = inPort.get(f);
			if (inBoundFromSrc.size() == 0) {
				ready = false;
			}
		}
		return ready;
	}

}
```

HeadBind.java

```
package run;
import java.util.AbstractList;

/* (1, 2)  "B"  true */
public class HeadBind {
	AbstractList<TailBind> binds;
	String cellLabel;
	boolean done;

	public boolean isReady(){
		return false;
	}
}
```

InitialFacet.java

```
package run;
import java.util.ArrayList;
import java.util.List;

import run.core.NIL;

public class InitialFacet extends Facet {

	private String value;
	private boolean sent = false;

	public InitialFacet(String name, Cell owner, boolean isInputFacet, String value) {
		super(name, owner, false);
		this.value = value;
	}

	@Override
	public List<Object> dispatchOutbound() {
```

```
                List<Object> outbound = new ArrayList<Object>();

                if (! sent)
                        outbound.add(Integer.parseInt(value));
                else
                        outbound.add(new NIL());
                sent = true;

                return outbound;
        }
}
```

IntFacet.java

```
package run;

import java.util.ArrayList;
import java.util.List;

public class IntFacet extends Facet {

        private String value;

        public IntFacet(String name, Cell owner, boolean isInputFacet, String value) {
                super(name, owner, false);
                this.value = value;
        }

        @Override
        public List<Object> dispatchOutbound() {
                List<Object> outbound = new ArrayList<Object>();
                outbound.add(Integer.parseInt(value));
                return outbound;
        }
}
```

Lattice.java

```
package run;

import java.util.ArrayList;
import java.util.HashSet;
import java.util.List;
import java.util.Set;

/**
 * A more refined object representing the interior structure of a cell.
 * @author Thomas Chau
 *
 */
public class Lattice {

	private ArrayList<Bind> bindings = new ArrayList<Bind>();
	private Set<Cell> subcells = new HashSet<Cell>();

	private String name;

	public Lattice(String name) {
		this.name = name;
	}

	//	public void addBind(run.client.Bindable lb, BindType bindType, run.client.Bindable
rb) {
	public void addBind(run.Bind b) {
			bindings.add(b);

			Cell leftOwner = b.getLeft().getOwner();
			Cell rightOwner = b.getRight().getOwner();

			subcells.add(leftOwner);
			subcells.add(rightOwner);
	}

	public ArrayList<Bind> getBindings() {
		return bindings;
	}

	public String toString() {
		String s = "";
		for (Bind b : bindings) {
			s += b.toString() + "\n";
		}
		return s;
	}

	public Set<Cell> getSubcells() {
		return subcells;
	}

	/**
	 * All facets that are destined to receive from the given facet.
	 * @param stimmer
	 * @return
	 */
	public List<Facet> getDestinations(Facet stimmer) {
		List<Facet> dests = new ArrayList<Facet>();
		for (Bind b : bindings) {
			if (b.getLeft() == stimmer)
				dests.add(b.getRight());
		}
		return dests;
	}
}
```

StrFacet.java

```
package run;

import java.util.ArrayList;
import java.util.List;

public class StrFacet extends Facet {

	private String origValue;
	private String value;

	public StrFacet(String name, Cell owner, boolean isInputFacet, String value) {
		super(name, owner, false);
		origValue = value;
		this.value = value;
	}

	@Override
	public List<Object> dispatchOutbound() {
		List<Object> outbound = new ArrayList<Object>();
		outbound.add(value);
		value = origValue;
		return outbound;
	}

	public void concat(Object concat) {
		value = value.concat(concat.toString());
	}
}
```

Built-in and Special Cells

ADD.java

```
package run.core;

import java.util.ArrayList;
import java.util.List;

import run.Cell;
import run.Facet;

/*
	ADD takes in n > 0 arguments and adds them. The individual
	input facets of ADD cannot be accessed individually, due to the nature of the cell.
		<(*) ADD (.out)>
		*: any number greater than 0 of INT inputs
		out: an INT
*/

public class ADD extends Cell {
	private Facet in, out;

	public ADD(String labelName) {
		super("ADD", labelName);
		in = new Facet(".in", this, INPUT_FACET);
		out = new Facet(".out", this, OUTPUT_FACET);
		out.addSource(in);
	}

	@Override
	public List<Facet> stimulate(Facet stimmer) {
		List<Object> summands = in.dispatchOutboundEntire();

		// CHECK FOR ALLNILS -- THEN MAKE NIL
		Integer sum = 0;
		boolean allNil = true;
		for (Object o : summands) {
			if (!(o instanceof NIL)) {
				allNil = false;
				Integer i = (Integer) o;
				sum += i;
			}
		}

		List<Object> sums = new ArrayList<Object>();

		if (allNil) {
			sums.add(new NIL());
		}
		else {
			sums.add(sum);
		}

		out.receive(in, sums);

		ArrayList<Facet> doneFacets = new ArrayList<Facet>();
		doneFacets.add(out);

		return doneFacets;
	}

	public List<Facet> getFacets()
	{
		ArrayList<Facet> temp = new ArrayList<Facet>();
		temp.add(in);
		temp.add(out);
		return temp;
	}
```

```
	//Dot printout method
	public String toDot(String prefix)
	{
		String returnString = "";
		returnString += "subgraph " + "cluster_" + cellTypeName + hashCode() + "{\n";
		returnString += "label = \"" + cellTypeName + "\";\n";

		//Print my facets
		returnString +=  prefix + cellTypeName + hashCode() + "in [label=\"in\"];\n";
		returnString += prefix + cellTypeName + hashCode() + "out [label=\"out\"];\n";

		returnString += "}\n";
		return returnString;
	}
}
```

EQUALS.java

```
package run.core;

import java.util.ArrayList;
import java.util.List;

import run.Cell;
import run.Facet;
/**
 * EQUALS cell
 *
 * @author Rajesh
 */
public class EQUALS extends Cell {
	private Facet left, right, output;
	private ArrayList<Object> internalValue;
	private boolean noInput;

	private Integer leftOp = null;
	private Integer rightOp = null;

	public EQUALS(String labelName) {
		super("EQUALS", labelName);
		left = new Facet(".left", this, INPUT_FACET);
		right = new Facet(".right", this, INPUT_FACET);
		output = new Facet(".output", this, OUTPUT_FACET);
		output.addSource(left);
		noInput = false;
	}

	@Override
	public List<Facet> stimulate(Facet stimmer)
	{
//		System.out.println("EQUALS");
		ArrayList<Facet> doneFacets = new ArrayList<Facet>();
		List<Object> op = stimmer.dispatchOutbound();

		// pass along NILS
		if (op.get(0) instanceof NIL) {
			output.receive(left, op);
			doneFacets.add(output);
			return doneFacets;
		}


		if (stimmer == left)
			leftOp = (Integer) op.get(0);
		else
			rightOp = (Integer) op.get(0);

		if (leftOp != null && rightOp != null) {

			List<Object> returns = new ArrayList<Object>();
			if (leftOp.compareTo(rightOp) == 0)
```

```
                        returns.add(new Integer(1));
                else
                        returns.add(new NIL());

                output.receive(left, returns);

                doneFacets.add(output);
                leftOp = rightOp = null;
        }
        return doneFacets;

    }

    public List<Facet> getFacets()
    {
        ArrayList<Facet> temp = new ArrayList<Facet>();

        if(noInput)
        {
            return temp;
        }

        temp.add(left);
        temp.add(right);
        temp.add(output);
        return temp;
    }

    //Dot printout method
    public String toDot(String prefix)
    {
        String returnString = "";
        returnString += "subgraph " + "cluster_" + cellTypeName + hashCode() + "{\n";
        returnString += "label = \"" + cellTypeName + "\";\n";

        //Print my facets
        returnString +=  prefix + cellTypeName + hashCode() + "left
[label=\".left\"];\n";
        returnString += prefix + cellTypeName + hashCode() + "right
[label=\".right\"];\n";
        returnString += prefix + cellTypeName + hashCode() + "output
[label=\".output\"];\n";

        returnString += "}\n";
        return returnString;
    }
}
```

GREATER_THAN.java

```
package run.core;

import java.util.ArrayList;
import java.util.List;

import run.Cell;
import run.Facet;
import run.Lattice;

public class GREATER_THAN extends Cell {
	private Facet left, right, output;
	private boolean noInput;
	private Integer rightOp;
	private Integer leftOp;

	public GREATER_THAN(String labelName) {
		super("GREATER_THAN", labelName);
		left = new Facet(".left", this, INPUT_FACET);
		right = new Facet(".right", this, INPUT_FACET);
		output = new Facet(".output", this, OUTPUT_FACET);
		output.addSource(left);
		noInput = false;
	}

	@SuppressWarnings("unchecked")
	@Override
	public List<Facet> stimulate(Facet stimmer)
	{
		ArrayList<Facet> doneFacets = new ArrayList<Facet>();
		List<Object> op = stimmer.dispatchOutbound();

		// pass along NILS
		if (op.get(0) instanceof NIL) {
			output.receive(left, op);
			doneFacets.add(output);
			return doneFacets;
		}

		if (stimmer == left)
			leftOp = (Integer) op.get(0);
		else
			rightOp = (Integer) op.get(0);

		if (leftOp != null && rightOp != null) {

			List<Object> returns = new ArrayList<Object>();
			if (leftOp.intValue() > rightOp.intValue())
				returns.add(new Integer(1));
			else
				returns.add(new NIL());

			output.receive(left, returns);

			doneFacets.add(output);
			leftOp = rightOp = null;
		}
		return doneFacets;
	}
	public List<Facet> getFacets()
	{
		ArrayList<Facet> temp = new ArrayList<Facet>();
		temp.add(left);
		temp.add(right);

		if(noInput)
		{
			return temp;
		}

		temp.add(output);
		return temp;
```

```
        }

        public void secrete()
        {
        }

        //Dot printout method
        public String toDot(String prefix)
        {
                String returnString = "";
                returnString += "subgraph " + "cluster_" + cellTypeName + hashCode() + "{\n";
                returnString += "label = \"" + cellTypeName + "\";\n";

                //Print my facets
                returnString +=  prefix + cellTypeName + hashCode() + "left
[label=\".left\"];\n";
                returnString += prefix + cellTypeName + hashCode() + "right
[label=\".right\"];\n";
                returnString += prefix + cellTypeName + hashCode() + "output
[label=\".output\"];\n";

                returnString += "}\n";
                return returnString;
        }

}
```

GUI.java

```
package run.core;
import java.awt.Color;
import javax.swing.*;
import java.awt.*;
import java.util.ArrayList;
import java.util.List;

import run.Cell;
import run.Facet;
import run.StrFacet;

/** String Cell
 *
 * @author Rajesh.ramakrishnan
 */
public class GUI extends Cell {
        private Facet in;
        private StrFacet out;
        private String myValue;
        public static int[][] grid;
        public static int count = 0;
        public static JFrame frame;
        public static Container pane;
        public static int width = 600;
        public static int height = 600;

        public GUI(String name) {
                super("GUI", name);
                myValue = name;
                in = new Facet(".in", this, INPUT_FACET);
                out = new StrFacet(".out", this, OUTPUT_FACET, name);

                if(grid == null){
                        grid = new int[15][15];
                }

                if(frame == null){
                        frame = new JFrame();
                        frame.setSize(width, height);
                        frame.setTitle("Game of Life");
```

```
                        pane = frame.getContentPane();
                        pane.setLayout(null);
                        pane.setBackground(Color.PINK);
                }
        }

        /*@Override
        public void feedInput(Object data) {

        }*/

        /**
         * Hacked up to PRINT when stimulated.
         */
        @Override
        public List<Facet> stimulate(Facet stimmer)
        {
                ArrayList<Facet> doneFacets = new ArrayList<Facet>();
                List<Object> vals = in.dispatchOutbound();
//ignore nils

                Object concat = vals.get(0);
                if (!(concat instanceof NIL)) {
                        if(!myValue.equals("")){
                                count++;
                                String[] values = myValue.split(",");
                                grid[Integer.parseInt(values[0])][Integer.parseInt(values[1])] =
Integer.parseInt(concat.toString());
                                //System.out.println(myValue + ':' + concat);
                                out.concat(concat);
                                doneFacets.add(out);
                        }
                }
//              else System.out.println("NIL TO STR");
                if(count == ((15*15) - 1 )){
                        frame.setVisible(true);
                        //String output = "";
                        pane.removeAll();
                        for(int i = 0; i < grid.length; i++){
                                for(int j = 0; j < grid[i].length; j++){
                                        //output += grid[i][j];
                                        JPanel panel = new JPanel();
                                        panel.setBounds(j*40, i*40, 40, 40);
                                        pane.add(panel);
                                        if(grid[i][j] != 0){
                                                panel.setBackground(Color.GREEN);
                                        }
                                        else{
                                                panel.setBackground(Color.PINK);
                                        }
                                }
                                //output += "\n";
                        }
                        count = 0;
                        try{
                                Thread.sleep(200);    //slow down the animation
                        }
                        catch(Exception ex){}
                        pane.repaint();
                        frame.repaint();
                }
                return doneFacets;
        }

        protected String injectInt(String s, int val) { return s.replaceFirst("%d",
((Integer)val).toString()); }
        protected String inject(String s, Object o) { return s.replaceFirst("%s", o.toString());
}

        /** Find and fix special characters.
         *
```

```
	 * @author Rajesh.ramakrishnan
	 */
	protected List<Object> process(String nm) {
		ArrayList<Object> vals = new ArrayList<Object>();
		String tmp = nm.replaceAll("\\\\n", "\n");
		tmp = tmp.replaceAll("\\\\t", "\t");
		tmp = tmp.replaceAll("\\\\b", "\b");
		tmp = tmp.replaceAll("\\\\f", "\f");
		tmp = tmp.replaceAll("\\\\r", "\r");
		tmp = tmp.replaceAll("\\\\u", "\r");
		tmp = tmp.replaceAll("\\\\%", "%");
		vals.add(tmp);
		return vals;
	}

	//Dot printout method
	public String toDot()
	{
		String returnString = "";
		returnString += "subgraph " + hashCode() + "{\n";
		returnString += "label = \"" + cellTypeName + "\";\n";
		returnString += "}\n";
		return returnString;
	}


	public List<Facet> getFacets()
	{
		ArrayList<Facet> temp = new ArrayList<Facet>();
		temp.add(in);
		temp.add(out);
		return temp;
	}

	public List<Facet> getOutFacets() {
		ArrayList<Facet> arrayList = new ArrayList<Facet>();
		arrayList.add(out);
		return arrayList;
	}

	@Override
	public boolean isLiteral() {
		return false;
	}
}
```

INITIAL.java

```
package run.core;

import java.util.ArrayList;
import java.util.List;

import run.Cell;
import run.Facet;
import run.InitialFacet;

/**
 * Cell that propagates a value once and never again.
 * @author Thomas Chau
 *
 */
public class INITIAL extends Cell {

	private Facet value;

	public INITIAL(String valueLabel) {
		super("INITIAL", valueLabel);
		value = new InitialFacet(".value", this, OUTPUT_FACET, valueLabel);
	}

	@Override
	public List<Facet> stimulate(Facet stimmer)
	{
		return null;
	}

	//Dot printout method
	public String toDot(String prefix)
	{
		String returnString = "";
		returnString += "subgraph " + "cluster_" + cellTypeName + hashCode() + "{\n";
		returnString += "label = \"" + labelName + "\";\n";

		//Print my facets
		returnString += prefix + cellTypeName + hashCode() + "value
[label=\".value\"];\n";

		returnString += "}\n";
		return returnString;
	}

	public List<Facet> getFacets()
	{
		ArrayList<Facet> temp = new ArrayList<Facet>();
		temp.add(value);
		return temp;
	}

	/**
	 * Returns .value facet with preloaded number.
	 */
	public List<Facet> getOutFacets() {
		ArrayList<Facet> arrayList = new ArrayList<Facet>();
		arrayList.add(value);
		return arrayList;

	}

	public boolean isLiteral() {
		return true;
	}
}
```

INT.java

```
package run.core;

import java.util.ArrayList;
import java.util.List;

import run.Cell;
import run.Facet;
import run.IntFacet;

public class INT extends Cell {

	private Facet value;
	private Facet phantom;

	public INT(String valueLabel) {
		super("INT", valueLabel);
		value = new IntFacet(".value", this, OUTPUT_FACET, valueLabel);
	}

	@Override
	public List<Facet> stimulate(Facet stimmer)
	{
		return null;
	}

	public void secrete()
	{
	}

	//Dot printout method
	public String toDot(String prefix)
	{
		String returnString = "";
		returnString += "subgraph " + "cluster_" + cellTypeName + hashCode() + "{\n";
		returnString += "label = \"" + labelName + "\";\n";

		//Print my facets
		returnString += prefix + cellTypeName + hashCode() + "value
[label=\".value\"];\n";

		returnString += "}\n";
		return returnString;
	}

	public List<Facet> getFacets()
	{
		ArrayList<Facet> temp = new ArrayList<Facet>();
		temp.add(phantom);
		temp.add(value);
		return temp;
	}

	/**
	 * Returns .value facet with preloaded number.
	 */
	public List<Facet> getOutFacets() {
		ArrayList<Facet> arrayList = new ArrayList<Facet>();
		arrayList.add(value);
		return arrayList;

	}

	public boolean isLiteral() {
		return true;
	}
}
```

LESS_THAN.java

```
package run.core;
```

```
import java.util.ArrayList;
import java.util.List;

import run.Cell;
import run.Facet;
import run.Lattice;

public class LESS_THAN extends Cell {
	private Facet left, right, output;
	private ArrayList<Object> internalValue;
	private boolean noInput;
	private Integer leftOp;
	private Integer rightOp;

	public LESS_THAN(String labelName) {
		super("LESS_THAN", labelName);
		left = new Facet(".left", this, INPUT_FACET);
		right = new Facet(".right", this, INPUT_FACET);
		output = new Facet(".output", this, OUTPUT_FACET);
		output.addSource(left);
		internalValue = new ArrayList<Object>();
		noInput = false;
	}

	@SuppressWarnings("unchecked")
	@Override
	public List<Facet> stimulate(Facet stimmer) {
//		System.out.println("LESS_THAN");
		ArrayList<Facet> doneFacets = new ArrayList<Facet>();
		List<Object> op = stimmer.dispatchOutbound();

		// pass along NILS
		if (op.get(0) instanceof NIL) {
			output.receive(left, op);
			doneFacets.add(output);
			return doneFacets;
		}


		if (stimmer == left)
			leftOp = (Integer) op.get(0);
		else
			rightOp = (Integer) op.get(0);

		if (leftOp != null && rightOp != null) {

			List<Object> returns = new ArrayList<Object>();
			if (leftOp.intValue() < rightOp.intValue())
				returns.add(new Integer(1));
			else
				returns.add(new NIL());

			output.receive(left, returns);

			doneFacets.add(output);
			leftOp = rightOp = null;
		}
		return doneFacets;
	}
	public List<Facet> getFacets()
	{
		ArrayList<Facet> temp = new ArrayList<Facet>();
		temp.add(left);
		temp.add(right);

		if(noInput)
		{
			return temp;
		}

		temp.add(output);
```

```
                return temp;
        }

        //Dot printout method
        public String toDot(String prefix)
        {
                String returnString = "";
                returnString += "subgraph " + "cluster_" + cellTypeName + hashCode() + "{\n";
                returnString += "label = \"" + cellTypeName + "\";\n";

                //Print my facets
                returnString +=  prefix + cellTypeName + hashCode() + "left
[label=\".left\"];\n";
                returnString += prefix + cellTypeName + hashCode() + "right
[label=\".right\"];\n";
                returnString += prefix + cellTypeName + hashCode() + "output
[label=\".output\"];\n";

                returnString += "}\n";
                return returnString;
        }

}
```

MEM.java

```
package run.core;

import java.util.ArrayList;
import java.util.List;

import run.Cell;
import run.Facet;

/**
 * Memory cell
 * May .set values
 * or .release values onto .val for exporting
 * or .init values to start off.
 * @author Thomas Chau
 *
 */
public class MEM extends Cell {

	// MEM is a state element -- this is the memory.
	private Object memory;
	private Facet set;
	private Facet release;
	private Facet val;
	private Facet init;
	private boolean onlySet;


	public MEM(String name) {
		super("MEM", name);
		val = new Facet(".val", this, false);
		set = new Facet(".set", this, true);
		init = new Facet(".init", this, true);
		release = new Facet(".release", this, true);
		val.addSource(set);

		onlySet = false;
	}

	/**
	 * Accept any .in and store
	 */
	@Override
	public List<Facet> stimulate(Facet stimmer) {

		ArrayList<Facet> doneFacets = new ArrayList<Facet>();

		// when receive a SET, anything not a NIL gets set as value, release.
		if (stimmer == set) {
			List<Object> incomingVals = set.dispatchOutbound();
			for (Object o : incomingVals)
				if (!(o instanceof NIL)) {
					memory = o;
					ArrayList<Object> memarr = new ArrayList<Object>();
					memarr.add(memory);
					val.receive(set, memarr);
					doneFacets.add(val);
				}
		}
		// when receive an .init, set the value inside
		else if (stimmer == init) {
			List<Object> initialValue = init.dispatchOutbound();
			if (!(initialValue.get(0) instanceof NIL)) {
				memory = initialValue.get(0);
				ArrayList<Object> memarr = new ArrayList<Object>();
				memarr.add(memory);
				val.receive(set, memarr);
				doneFacets.add(val);
			}
		}
```

```
                // when receive a .release, put out.
                else if (stimmer == release) {
                        List<Object> release_hits = release.dispatchOutbound();
                        Object arg = release_hits.get(0);
                        ArrayList<Object> memarr = new ArrayList<Object>();
                        if (!(arg instanceof NIL)) {
                                memarr.add(memory);
                        }
                        else {
                                memarr.add(arg);
                        }
                        val.receive(set, memarr);
                        doneFacets.add(val);
                }


                return doneFacets;
        }

        public List<Facet> getFacets()
        {
                ArrayList<Facet> temp = new ArrayList<Facet>();
                temp.add(set);
                if(!onlySet)
                {
                        temp.add(val);
                }
                temp.add(release);
                temp.add(init);
                return temp;
        }

        //Dot printout method
        public String toDot(String prefix)
        {
                String returnString = "";
                returnString += "subgraph " + "cluster_" + cellTypeName + hashCode() + "{\n";
                returnString += "label = \"" + cellTypeName + "\";\n";

                //Print my facets
                returnString += prefix + cellTypeName + hashCode() + "val [label=\".val\"];\n";
                returnString +=  prefix + cellTypeName + hashCode() + "set [label=\".set\"];\n";
                returnString += prefix + cellTypeName + hashCode() + "release
[label=\".release\"];\n";

                returnString += "}\n";
                return returnString;
        }
}
```

NEGATE.java

```
package run.core;

import java.util.ArrayList;
import java.util.List;

import run.Cell;
import run.Facet;

/*
Negates booleans (represented as 1 or 0
*/

public class NEGATE extends Cell {
	private Facet in, out;
	private ArrayList<Object> internalValue;

	public NEGATE(String labelName) {
		super("NEGATE", labelName);
		in = new Facet(".in", this, INPUT_FACET);
		out = new Facet(".out", this, OUTPUT_FACET);
		out.addSource(in);
	}

	@Override
	public List<Facet> stimulate(Facet stimmer) {
		ArrayList<Facet> doneFacets = new ArrayList<Facet>();

		List<Object> arg = in.dispatchOutbound();
		List<Object> vals = new ArrayList<Object>();

		// IGNORE NILS
		Object argo = arg.get(0);
		if (argo instanceof NIL) {
//			System.out.println("   NEGATE : nil");
			vals.add(argo);
			out.receive(in, vals);
			doneFacets.add(out);
			return doneFacets;
		}
		else {
			Integer i = (Integer) arg.get(0);

			if (i.intValue() == 0) {
				vals.add(new Integer(1));
				out.receive(in, vals);
			}
			else if (i.intValue() == 1) {
				vals.add(new Integer(0));
				out.receive(in, vals);
			}

			doneFacets.add(out);
			return doneFacets;
		}
	}

	public List<Facet> getFacets()
	{
		ArrayList<Facet> temp = new ArrayList<Facet>();
		temp.add(in);
		temp.add(out);
		return temp;
	}

	//Dot printout method
	public String toDot(String prefix)
	{
		String returnString = "";
		returnString += "subgraph " + "cluster_" + cellTypeName + hashCode() + "{\n";
		returnString += "label = \"" + cellTypeName + "\";\n";
```

```
                //Print my facets
                returnString +=  prefix + cellTypeName + hashCode() + "in [label=\"in\"];\n";
                returnString += prefix + cellTypeName + hashCode() + "out [label=\"out\"];\n";

                returnString += "}\n";
                return returnString;
        }

}
```

NIL.java

```
package run.core;

/**
 * Represents dead branches. Passed along when a channel should be ignored.
 * @author Thomas Chau
 *
 */
public class NIL {

        int NIL_CODE = -1111;
        public int hashCode() {
                return -1111;
        }

        public String toString() {
                return "NIL";
        }
        @Override
        public boolean equals(Object other) {
                return (other instanceof NIL);
        }
}
```

SOUT.java

```
package run.core;

import java.util.ArrayList;
import java.util.List;

import run.Cell;
import run.Facet;

/**
        SOUT takes in an ArrayList of objects and prints them out onto standard output as comma-
separated
        strings.
                <(.data) SOUT>
                data: a list of INT, FLT, BOOL, or STR
        @author Greg Bramble
*/

public class SOUT extends Cell {
        Facet data;

        public SOUT(String labelName){
                super("SOUT", labelName);
                data = new Facet(".data", this, INPUT_FACET);
        }

        //converts the value of the facet to an ArrayList of objects and prints each one
separated by a comma
        @Override
        public List<Facet> stimulate(Facet stimmer) {
                List<Object> datums = data.dispatchOutbound();
```

```
                for (Object o : datums)
                        if (!(o instanceof NIL))
                                System.out.println(o);

                System.out.println("");
                return new ArrayList<Facet>();
        }

        public List<Facet> getFacets()
        {
                ArrayList<Facet> temp = new ArrayList<Facet>();
                temp.add(data);
                return temp;
        }

        //Dot printout method
        public String toDot(String prefix)
        {
                String returnString = "";
                returnString += "subgraph " + "cluster_" + cellTypeName + hashCode() + "{\n";
                returnString += "label = \"" + cellTypeName + "\";\n";

                //Print my facets
                returnString += prefix + cellTypeName + hashCode() + "data [label=\".data\"];\n";

                returnString += "}\n";
                return returnString;
        }
}
```

StrCell.java

```
package run.core;
import java.util.ArrayList;
import java.util.List;

import run.Cell;
import run.Facet;
import run.StrFacet;

/** String Cell
 *
 * @author Rajesh.ramakrishnan
 */
public class StrCell extends Cell {
	private Facet in;
	private StrFacet out;
	private String myValue;
	public StrCell(String name) {
		super("STR", name);
		myValue = name;
		in = new Facet(".in", this, INPUT_FACET);
		out = new StrFacet(".out", this, OUTPUT_FACET, name);
	}

	/*@Override
	public void feedInput(Object data) {

	}*/

	/**
	 * Hacked up to PRINT when stimulated.
	 */
	@Override
	public List<Facet> stimulate(Facet stimmer)
	{
		ArrayList<Facet> doneFacets = new ArrayList<Facet>();
		List<Object> vals = in.dispatchOutbound();

		Object concat = vals.get(0);
		if (!(concat instanceof NIL)) {
			System.out.println(myValue + ":" + concat);
			out.concat(concat);
			doneFacets.add(out);
		}

		return doneFacets;
	}

	protected String injectInt(String s, int val) { return s.replaceFirst("%d",
((Integer)val).toString()); }
	protected String inject(String s, Object o) { return s.replaceFirst("%s", o.toString());
}

	/** Find and fix special characters.
	 *
	 * @author Rajesh.ramakrishnan
	 */
	protected List<Object> process(String nm) {
		ArrayList<Object> vals = new ArrayList<Object>();
		String tmp = nm.replaceAll("\\\\n", "\n");
		tmp = tmp.replaceAll("\\\\t", "\t");
		tmp = tmp.replaceAll("\\\\b", "\b");
		tmp = tmp.replaceAll("\\\\f", "\f");
		tmp = tmp.replaceAll("\\\\r", "\r");
		tmp = tmp.replaceAll("\\\\u", "\r");
		tmp = tmp.replaceAll("\\\\%", "%");
		vals.add(tmp);
		return vals;
	}
```

```
	//Dot printout method
	public String toDot()
	{
		String returnString = "";
		returnString += "subgraph " + hashCode() + "{\n";
		returnString += "label = \"" + cellTypeName + "\";\n";
		returnString += "}\n";
		return returnString;
	}


	public List<Facet> getFacets()
	{
		ArrayList<Facet> temp = new ArrayList<Facet>();
		temp.add(in);
		temp.add(out);
		return temp;
	}

	public List<Facet> getOutFacets() {
		ArrayList<Facet> arrayList = new ArrayList<Facet>();
		arrayList.add(out);
		return arrayList;
	}

	@Override
	public boolean isLiteral() {
		return false;
	}
}
```

STRLIT.java

```
package run.core;
import java.util.ArrayList;
import java.util.List;

import run.Cell;
import run.Facet;
import run.StrFacet;

/** String Cell
 *
 * @author Rajesh.ramakrishnan
 */
public class STRLIT extends Cell {
	private Facet in;
	private StrFacet out;
	private String myValue;
	public STRLIT(String name) {
		super("STR", name);
		myValue = name;
		in = new Facet(".in", this, INPUT_FACET);
		out = new StrFacet(".out", this, OUTPUT_FACET, name);
	}

	@Override
	public List<Facet> stimulate(Facet stimmer)
	{
		ArrayList<Facet> doneFacets = new ArrayList<Facet>();
		List<Object> vals = in.dispatchOutbound();

		Object concat = vals.get(0);
		if (!(concat instanceof NIL)) {
			System.out.println(myValue + ":" + concat);
			out.concat(concat);
			doneFacets.add(out);
		}

		return doneFacets;
	}
```

```
	protected String injectInt(String s, int val) { return s.replaceFirst("%d",
((Integer)val).toString()); }
	protected String inject(String s, Object o) { return s.replaceFirst("%s", o.toString());
}

	/** Find and fix special characters.
	 *
	 * @author Rajesh.ramakrishnan
	 */
	protected List<Object> process(String nm) {
		ArrayList<Object> vals = new ArrayList<Object>();
		String tmp = nm.replaceAll("\\\\n", "\n");
		tmp = tmp.replaceAll("\\\\t", "\t");
		tmp = tmp.replaceAll("\\\\b", "\b");
		tmp = tmp.replaceAll("\\\\f", "\f");
		tmp = tmp.replaceAll("\\\\r", "\r");
		tmp = tmp.replaceAll("\\\\u", "\r");
		tmp = tmp.replaceAll("\\\\%", "%");
		vals.add(tmp);
		return vals;
	}

	//Dot printout method
	public String toDot()
	{
		String returnString = "";
		returnString += "subgraph " + hashCode() + "{\n";
		returnString += "label = \"" + cellTypeName + "\";\n";
		returnString += "}\n";
		return returnString;
	}


	public List<Facet> getFacets()
	{
		ArrayList<Facet> temp = new ArrayList<Facet>();
		temp.add(in);
		temp.add(out);
		return temp;
	}

	public List<Facet> getOutFacets() {
		ArrayList<Facet> arrayList = new ArrayList<Facet>();
		arrayList.add(out);
		return arrayList;
	}

	@Override
	public boolean isLiteral() {
		return true;
	}
}
```

Intermediate Representation Files

BindType.java

```
package ir;

import java.util.ArrayList;
import java.util.List;

import run.Bind;
import run.Facet;

/**
 * Bind Types (binding operators execute, linking facets)
 * @author Thomas Chau
 *
 */
public class BindType {
	private String bindType;

	public BindType(String bindType) {
		this.bindType = bindType;
	}

	public List<Bind> bind(List<Facet> leftBindables, List<Facet> rightBindables) throws
Exception {
		List<Bind> bindings = new ArrayList<Bind>();

		if (rightBindables.size() < 1) {
			throw new Exception("BINDTYPE::FAILED: BINDING COULD NOT FIND RIGHT
BINDABLES. Please specify the facets.");
		}
		if (leftBindables.size() < 1) {
			throw new Exception("BINDTYPE::FAILED: BINDING COULD NOT FIND LEFT
BINDABLES. Please specify the facets. SPECIFICALLY.");
		}

		if (bindType.equals("->")) {

			for (Facet left : leftBindables)
				for (Facet right : rightBindables)
					bindings.add(new Bind(left, this, right));

		}
		else if (bindType.equals("=>")) {
			int i = 0;
			for (Facet left : leftBindables) {
				Facet right = rightBindables.get(i);
				bindings.add(new Bind(left, this, right));
				i++;
			}

		}
		else {
			System.out.println("Bind failed: incorrect or unsupported bind type.");
			(new Exception()).printStackTrace();
		}

		return bindings;
	}

	public String toString() {
		return bindType;
	}
}
```

Block.java

```
package ir;
```

```
import java.util.ArrayList;
import java.util.List;

import run.Facet;

/**
 * IR Class representing Blocks in binding relationships in the statements.
 * For example:
 *
 * .a                                        BLOCK.ONE_FACET
 * SQUARE                                    BLOCK.UNIT (UNIT.GENERIC)
 * .in(SOMECELL).out          BLOCK.UNIT_FACET
 * (.a, .b)                                  BLOCK.FACETS
 * INT:2
 * .left(LESS_THAN:lt)
 * (.left(LESS_THAN:lt), .left(GREATER_THAN:gt)
 *
 * among others.
 *
 * @author Thomas Chau
 *
 */
public abstract class Block  {

	protected String type;

	protected ArrayList<CellUnit> units = new ArrayList<CellUnit>();
	protected ArrayList<Block> subblocks = new ArrayList<Block>();

	public void add(CellUnit u) {
		units.add(u);
	}

	public void add(Block b) {
		subblocks.add(b);
	}

	public void setType(String blockType) {
		this.type = blockType;
	}

	/**
	 * Execution-stage method to retrieve the Block's left-eligible facets.
	 * Construct any cells necessary.
	 * @param structuralDefinition
	 * @return
	 */
	public abstract List<Facet> getLeftFacets(StructuralDefinition structuralDefinition)
throws Exception;

	/**
	 * Execution-stage method to retrieve the Block's right-eligible facets.
	 * Construct any cells necessary.
	 * @param structuralDefinition
	 * @return
	 */
	public abstract List<Facet> getRightFacets(StructuralDefinition structuralDefinition)
throws Exception;

	/**
	 * Performs a Symbol-Table like seek of a cell instance.
	 * @param labelName
	 * @return
	 */
	public CellUnit findInstance(String labelName) {

		for (CellUnit b : units) {
			if ( !(b instanceof InstanceRef) && labelName.equals(b.getLabel()))
				return b;
		}
```

```
                CellUnit candidate = null;
                for (Block b : subblocks) {
                        candidate = b.findInstance(labelName);
                        if (candidate != null)
                                return candidate;
                }

                return candidate;
        }

        public String toString() {
                String s = "";
                for (CellUnit u : units)
                        s += u.toString() + ",";
                for (Block b : subblocks)
                        s += b.toString() + ",";
                return  s.substring(0, s.length() -1) ;
        }

        public Block getFirstSubBlock() {
                if (subblocks.size() == 1)
                        return subblocks.get(0);
                else return null;
        }

        /** All blocks need to clone
         */
        public abstract Block clone();
}
```

CellDef.java

```
package ir;

import run.Cell;
import run.Lattice;

/**
 * The generalized Cell. The Cell is responsible for managing bindings, facets,
 * and intracellular transportation of data subcells.
 *
 * User-defined cells "speciate" from this stem cell by providing their own
 * Structural Definition (a blueprint) that is realized into an actual Lattice of
 * subcells that govern the cell's behavior.
 *
 * All Cells perform a computation when stimulated on their (satisfied) facets.
 *
 * Cells manage the set of facets that they own and manage the flow of computation
 * through their internal subcell structure -- the lattice.
 *
 *
 * @author Thomas Chau
 *
 */
public class CellDef {

        protected String cellTypeName;
        protected StructuralDefinition d;

        public CellDef(String name) {
                this.cellTypeName = name;
        }

        public void setStructure(StructuralDefinition d) {
                this.d = d;
        }

        /**
         * Constructs the Cellular structure and returns reference to it.
         */
```

```
        public Cell realize() {
//              System.out.println("CellDef: Lattice of " + cellTypeName + " realizing...");
                return d.clone().realize(cellTypeName);
        }

        public String getName() {
                return cellTypeName;
        }
}
```

CellLib.java

```
package ir;

import java.util.HashMap;

public class CellLib {

        private static CellLib lmk;
        private HashMap<String, CellDef> dict = new HashMap<String, CellDef>();

        protected CellLib() {}

        public static CellLib getInstance() {
           if(lmk == null) {
              lmk = new CellLib();
           }
           return lmk;
        }

        public void addCellDef(CellDef d) {
                dict.put(d.getName(), d);
        }

        public CellDef getCellDef(String cellTypeName) {
                return dict.get(cellTypeName);
        }
}
```

CellUnit.java

```
package ir;

import run.Cell;
import run.core.INITIAL;
import run.core.INT;
import run.core.NEGATE;
import run.core.STRLIT;
import run.core.StrCell;

/**
 * An IR class representing the Cell unit.
 * @author Thomas Chau
 *
 */
public class CellUnit {

        // the type of cell
        protected String cellTypeName;

        // the label (if any) used by programmer for this instance
        protected String labelName;

        // the actual object in memory that this IR is supposed to resolve to
        private Cell underlying_cell;

        public CellUnit(String typeName) {
                cellTypeName = typeName;
                labelName = null;
        }
```

```
	public Cell genesis(StructuralDefinition d) {
		if (cellTypeName.equals("STR"))
			underlying_cell = new StrCell(labelName);
		else if (cellTypeName.equals("INT"))
			underlying_cell = new INT(labelName);
		else if (cellTypeName.equals("INITIAL"))
			underlying_cell = new INITIAL(labelName);
		else if (cellTypeName.equals("STRLIT"))
			underlying_cell = new STRLIT(labelName);
		else if (cellTypeName.equals("NEGATE"))
			underlying_cell = new NEGATE(labelName);
		else if (cellTypeName.equals("MEM"))
			underlying_cell = new run.core.MEM(labelName);
		else if (cellTypeName.equals("GREATER_THAN"))
			underlying_cell = new run.core.GREATER_THAN(labelName);
		else if (cellTypeName.equals("LESS_THAN"))
			underlying_cell = new run.core.LESS_THAN(labelName);
		else if (cellTypeName.equals("EQUALS"))
			underlying_cell = new run.core.EQUALS(labelName);
		else if (cellTypeName.equals("ADD"))
			underlying_cell = new run.core.ADD(labelName);
		else if (cellTypeName.equals("SOUT"))
			underlying_cell = new run.core.SOUT(labelName);
		else if (cellTypeName.equals("GUI"))
			underlying_cell = new run.core.GUI(labelName);
		else
			try { underlying_cell = (Cell)Class.forName("run.core." +
cellTypeName).newInstance(); }
			catch (Exception e) { underlying_cell = null; }

			// else, may be a user cell
		if (underlying_cell == null) {
			CellDef cellDef = CellLib.getInstance().getCellDef(cellTypeName);
			underlying_cell = cellDef.realize();
			underlying_cell.setLabelName(labelName);
			if (underlying_cell == null) {
				System.out.println("ERROR: UNIMPLEMENTED CELL: " + cellTypeName);
			}
		}

		return underlying_cell;
	}

	/**
	 * Label alias used by programmer.
	 * @param label
	 */
	public void setLabelName(String label) {
		labelName = label;
	}

	/**
	 * This is where you instantiate stuff such as your core cells,
	 * e.g. return new Sout();
	 * in the case that the cellTypeName is 'Sout'
	 */
	public Cell getUnderlyingObject(StructuralDefinition d) {
		if (underlying_cell == null) {
			underlying_cell = genesis(d);
		}

		return underlying_cell;
	}

	public String getLabel() {
		return labelName;
	}

	public String toString() {
		if (labelName != null)
```

```
                        return labelName;
                return cellTypeName;
        }

        public CellUnit clone() {
                CellUnit cellUnit = new CellUnit(cellTypeName);
                cellUnit.setLabelName(labelName);
                return cellUnit;
        }

}
```

InstanceRef.java

```
package ir;

import run.Cell;

public class InstanceRef extends CellUnit {

        public InstanceRef(String label) {
                super(label);
                setLabelName(label);
        }

        @Override
        public Cell getUnderlyingObject(StructuralDefinition d) {
                return d.findInstance(labelName).getUnderlyingObject(d);
        }

        /**
         * Resolve the reference (another cell instance defined earlier with in the same
         * structural definition.
         */
        public Cell genesis(StructuralDefinition d) {
                CellUnit instance = d.findInstance(getLabel());
                return instance.getUnderlyingObject(d);
        }

        public InstanceRef clone() {
                InstanceRef instanceRef = new InstanceRef(cellTypeName);
                instanceRef.setLabelName(cellTypeName);
                return instanceRef;
        }

}
```

Stmt.java

```
package ir;

import ir.blocks.UnitFacetBlk;
import ir.blocks.UnitListsBlk;

import java.util.ArrayList;
import java.util.List;

import run.Bind;
import run.Facet;

/**
 * An IR cell representing "chains" of bindings.
 * @author Thomas Chau
 *
 */
public class Stmt {

        private ArrayList<Bind> bindings;
        private ArrayList<Object> block_bind_list; // unrefined list of blocks and binds to be
later processed
```

```
	/**
	 * The general Statement: a chain of bindings.
	 * BLK_1 -> BLK_2
	 * Binds the right-operands of BLK_1 to the left-operands of BLK_2
	 *		e.g.
	 *			CELLA(.a,.b) -> .c(CELLB)
	 *		binds .a,.b to .c
	 * @param list
	 */
	public Stmt(ArrayList<Object> list) {
		assert ((list.size() % 2) != 0);
		this.block_bind_list = list;
	}

	/**
	 * BLK -> BLK -> BLK -> BLK
	 * Returns the list of bindings represented.
	 * @return list of bindings represented by this statement
	 */
	public ArrayList<run.Bind> getBinds(StructuralDefinition d) {

			bindings = new ArrayList<Bind>();
			ArrayList<Object> list = block_bind_list;

			for (int i = 1; i < list.size() -1; i += 2) {
				try {
				// for each odd-numbered entry (a bind) add children, enter it
				ir.Block leftblock = (ir.Block) list.get(i - 1);
				List<Facet> left_operands = leftblock.getRightFacets(d);

				ir.Block rightblock = (ir.Block) list.get(i + 1);
				List<Facet> right_operands = rightblock.getLeftFacets(d);

				// now link them
				BindType bindType = (BindType) list.get(i);

				// localized, block-level bindings
				List<Bind> local_bindings = null;

					local_bindings = bindType.bind(left_operands,
right_operands);


				// added to the whole statement's bindings total
				bindings.addAll(local_bindings);
				} catch (Exception e) {
					e.printStackTrace();
					while(true){}
				}
			}
			return bindings;
	}

	public CellUnit findInstance(String labelName) {
		for (Object o : block_bind_list) {
			if (o instanceof Block) {
				CellUnit candidate = ((Block) o).findInstance(labelName);
				if (candidate != null)
					return candidate;
			}
		}
		return null;
	}

	public String toString() {
		String s = "";
		for (Object o : block_bind_list) {
			s += o.toString();
		}
		return s;
```

```
	}

	public void preFilter() {
		// preprocess the blocks
		for (Object o : block_bind_list) {
			if (o instanceof UnitListsBlk) {
				UnitListsBlk b = (UnitListsBlk) o;

				UnitFacetBlk unitfacet = b.xferUnitFacet();
				if (unitfacet != null) {
					block_bind_list.set(block_bind_list.indexOf(o), unitfacet);
				}

			}
		}
	}

	public Stmt clone() {
		ArrayList<Object> innard = new ArrayList<Object>();
		for (Object o : block_bind_list) {
			if (o instanceof Block)
				innard.add(((Block)o).clone());
			else innard.add(o);
		}
		return new Stmt(innard);
	}
}
```

StructuralDefinition.java

```
package ir;

import java.util.ArrayList;
import java.util.List;

import run.Bind;
import run.Cell;
import run.Facet;
import run.Lattice;

/**
 * A structural definition is an Intermediate Representation class which
 * maintains a "blue print", the definition, for a cell's internal structure.
 * Realizing the structure will produce a Lattice, which encodes
 * the structure of the cell.
 *
 * @author Thomas Chau
 *
 */
public class StructuralDefinition {

	/* Statements */
	public ArrayList<Stmt> statements;

	// Facets for this structure
	public ArrayList<Facet> ourFacets;
	// Cell of this structure, to be returned.
	public Cell myCell;

	public StructuralDefinition() {
		statements = new ArrayList<Stmt>();
		ourFacets = new ArrayList<Facet>();
	}

	/**
	 * Adds a statement's contribution to the structural definition.
	 * @param s
	 */
	public void add(Stmt s) {
		statements.add(s);
	}
```

```
	/**
	 * Realizes the structural definition, prior to execution.
	 * @param name
	 */
	public Cell realize(String name) {
		Lattice lat = new Lattice(name);
		myCell = new Cell(name, lat);

		for (Stmt s : statements)
			s.preFilter();

		// Retrieve the decomposed binds for each Statement.
		for(Stmt s : statements) {
			ArrayList<run.Bind> binds = s.getBinds(this);
			for (Bind b : binds) {
				lat.addBind(b);
			}
		}

		// REALIZE: here or in Stimulate()??????? broke.
		myCell.realize();
		return myCell;
	}

	/**
	 * In its symbolTable-like role, the definition will search and retrieve the
	 * CellUnit identified by this label.
	 * @param labelName
	 */
	public CellUnit findInstance(String labelName) {
		for (Stmt s : statements) {
			CellUnit candidate = s.findInstance(labelName);
			if (candidate != null)
				return (CellUnit) candidate;
		}
		return null;
	}

	/**
	 * Retrieves a Facet reference.
	 * If no such Facet exists already, constructs a new one and
	 * associates it with the Cell we are building.
	 * @param name
	 * @return
	 */
	public Facet getFacet(String name, boolean isInputFacet) {
		Facet newFacet;
		for (Facet f : ourFacets)
			if (f.getName().equals(name))
				return f;

		newFacet = new Facet(name, myCell, isInputFacet);
		ourFacets.add(newFacet);

		return newFacet;
	}

	public StructuralDefinition clone() {
		StructuralDefinition newDef = new StructuralDefinition();
		for (Stmt s : statements)
			newDef.add(s.clone());
		return newDef;
	}
}
```

FacetsBlk.java

```
package ir.blocks;

import ir.Block;
```

```
import ir.StructuralDefinition;

import java.util.LinkedList;
import java.util.List;

import run.Facet;

/**
 * A block representing a list of Facets
 * e.g.:
 *             (.a, .b, .c)
 * @author Thomas Chau
 *
 */
public class FacetsBlk extends Block {

	List<String> facetNames;
	List<Facet> realFacets;
	UnitBlk owner;

	public FacetsBlk(List<String> facets) {
		this.facetNames = facets;
	}

	/**
	 * Return facets of the owning structural definition.
	 */
	@Override
	public List<Facet> getLeftFacets(
			StructuralDefinition d) {
		LinkedList<Facet> facets = new LinkedList<Facet>();
		for (String name : facetNames) {
			Facet facet = d.getFacet(name, false);
			facets.add(facet);
		}

		return facets;
	}

	/**
	 * Return uncontextualized facets.
	 */
	@Override
	public List<Facet> getRightFacets(
			StructuralDefinition d) {
		LinkedList<Facet> facets = new LinkedList<Facet>();
		for (String name : facetNames) {
			Facet facet = d.getFacet(name, true);
			facets.add(facet);
		}

		return facets;
	}

	public String toString() {
		String s = "";
		for (String f : facetNames) {
			s += f + " ";
		}
		return s;

	}

	public void setOwner(UnitBlk owner) {
		this.owner = owner;
	}

	public boolean contains(String name) {
		return facetNames.contains(name);
	}
```

```
	@Override
	public FacetsBlk clone() {
		FacetsBlk facetsBlk = new FacetsBlk(facetNames);

		if (owner != null)
			facetsBlk.setOwner(owner.clone());
		return facetsBlk;
	}


}
```

UnitBlk.java

```
package ir.blocks;

import ir.Block;
import ir.CellUnit;
import ir.StructuralDefinition;

import java.util.ArrayList;
import java.util.List;

import run.Cell;
import run.Facet;

/**
 * A block representing a single unit:
 *      MYCELL
 * @author Thomas Chau
 *
 */
public class UnitBlk extends Block {

	private Cell cell;

	/**
	 * Get the Unit's facets eligible for Left. [X|]
	 */
	@Override
	public List<Facet> getLeftFacets(StructuralDefinition d) {
		if ( cell == null) {
			CellUnit cellUnit = units.get(0);
			cell = cellUnit.genesis(d);
		}

		return cell.getInFacets();
	}

	/**
	 * Get the unit's facets eligible for Right binding. [|X]
	 */
	@Override
	public List<Facet> getRightFacets(StructuralDefinition d) {
		if ( cell == null) {
			CellUnit cellUnit = units.get(0);
			cell = cellUnit.genesis(d);
		}

		return cell.getOutFacets();
	}

	// has only one unit
	public CellUnit getUnit() {
		return units.get(0);
	}

	public UnitBlk clone() {
		UnitBlk unitBlk = new UnitBlk();
		unitBlk.add(units.get(0).clone());
		return unitBlk;
```

```
	}
}
```

UnitFacetBlk.java

```
package ir.blocks;

import ir.Block;
import ir.CellUnit;
import ir.StructuralDefinition;

import java.util.LinkedList;
import java.util.List;

import run.Cell;
import run.Facet;

/**
 * A block representing contextualized facets:
 *             (.a,.b)(MYCELL)(.c,.d)
 * @author Thomas Chau
 *
 */
public class UnitFacetBlk extends Block{

	UnitBlk core;
	FacetsBlk left;
	FacetsBlk right;

	Cell coreReal;

	public UnitFacetBlk() {}
	public void setUnit(UnitBlk b) {
		core = b;

	}

	public UnitFacetBlk(UnitBlk ub) {
		core = ub;
	}

	public void setLeft(FacetsBlk b) {
		// give left a link to this unit, contextualize the Facets to the Cell
		left = b;
		left.setOwner(core);
	}

	public void setRight(FacetsBlk b) {
		right = b;
		right.setOwner(core);
	}

	/**
	 * Contextualize the facets on the LEFT to the Cell (our Unit). Return them.
	 *
	 * CASE: NONE -- will return ALL!!!!!!!
	 */
	@Override
	public List<Facet> getLeftFacets(
			StructuralDefinition d) throws Exception {

		// lazy instantiate.
		if (coreReal == null)
			coreReal = core.getUnit().genesis(d);

		// TODO: Tmp: disabled ->ADD-> defaulting behavior.
//		List<Facet> facets = coreReal.getInFacets();
		List<Facet> facets = coreReal.getFacets();

		if(left != null){
			boolean legal = false;
```

```
                for(String f: left.facetNames){
                        for(Facet fac: facets){
                                if(fac.getName().equals(f))
                                        legal = true;
                        }
                        if(!legal)
                                throw new Exception("The facet " + f + " is not legal.");
                        legal = false;
                }
        }
        List<Facet> selected = new LinkedList<Facet>();
        for (Facet f : facets) {
                if (left != null && left.contains(f.getName()))
                        selected.add(f);
        }
        // if the facets are unspecified, assume ALL FACETS ELIGIBLE FOR LEFT
        if (left == null) {
                selected = facets;
        }

        return selected;
}


/**
 * Contextualize the facets on the RIGHT to the Cell (our Unit). Return them.
 */
@Override
public List<Facet> getRightFacets(
                StructuralDefinition d) throws Exception{

        //      lazy instantiate.
        if (coreReal == null)
                coreReal = core.getUnit().genesis(d);

//              List<Facet> facets = coreReal.getOutFacets();
                List<Facet> facets = coreReal.getFacets();

                if(right != null){
                        boolean legal = false;
                        for(String f: right.facetNames){
                                for(Facet fac: facets){
                                        if(fac.getName().equals(f))
                                                legal = true;
                                }
                                if(!legal)
                                        throw new Exception("The facet " + f + " is not legal.");
                                legal = false;
                        }
                }
                List<Facet> selected = new LinkedList<Facet>();
                for (Facet f : facets) {
                        if (right != null && right.contains(f.getName()))
                                selected.add(f);
                }
                // if the facets are unspecified, assume ALL FACETS ELIGIBLE FOR RIGHT
                if (right == null)
                        selected = facets;


                return selected;
        }

        public String toString() {
                String s = "";
                if (left != null)
                        s += "(" + left.toString() + ")";
                s += "(" + core.toString() + ")";
                if (right != null)
                        s += "(" + right.toString() + ")";
                return s;
```

```
	}

	public CellUnit findInstance(String labelName) {
		try {
			if (core.getUnit().getLabel().equals(labelName))
				return core.getUnit();
			else return null;
		} catch (Exception e) {
			System.out.println("Failed trying to resolve instance of " + labelName);
		}
		return null;
	}

	public UnitFacetBlk clone() {
		UnitFacetBlk replace = new UnitFacetBlk();
		replace.setUnit(core.clone());

		if (left != null)
			replace.setLeft(left.clone());

		if (right != null)
			replace.setRight(right.clone());
		return replace;
	}
}
```

UnitListsBlk.java

```
package ir.blocks;

import ir.Block;
import ir.CellUnit;
import ir.StructuralDefinition;

import java.util.ArrayList;
import java.util.List;

import run.Cell;
import run.Facet;

/**
 * A block representing lists of Units.
 * @author Thomas Chau
 *
 */
public class UnitListsBlk extends Block {

	@Override
	public List<Facet> getLeftFacets(StructuralDefinition d) throws Exception{

		// return the left bind-eligible operands
		ArrayList<Facet> realizedUnits = new ArrayList<Facet>();

			for (ir.Block b : subblocks) {
				// here, grab facets for this D's cell .. add if not already
defined (fn in D);
				List<Facet> leftBindables = b.getLeftFacets(d);
				realizedUnits.addAll(leftBindables);
			}

			for (CellUnit c : units) {
				// resolve from the symbol table
				Cell cell = c.getUnderlyingObject(d);
				realizedUnits.addAll(cell.getInFacets());
			}

		return realizedUnits;
	}

	@Override
	public List<Facet> getRightFacets(StructuralDefinition d) throws Exception{
		// return the left bind-eligible operands
		ArrayList<Facet> realizedUnits = new ArrayList<Facet>();

			for (ir.Block b : subblocks) {
				// here, grab facets for this D's cell .. add if not already
defined (fn in D);
				List<Facet> rightBindables = b.getRightFacets(d);
				realizedUnits.addAll(rightBindables);
			}

			for (CellUnit c : units) {
				// resolve from the symbol table
				Cell cell = c.getUnderlyingObject(d);
				realizedUnits.addAll(cell.getOutFacets());
			}
//		}

		return realizedUnits;
	}

	protected UnitBlk getCore() {
		for (Block b : subblocks)
			if (b instanceof UnitBlk)
```

```
                        return (UnitBlk) b;
                return null;
        }

        // bad hack for AST correction
        public UnitFacetBlk xferUnitFacet() {
                if (type.equals("BLOCK.UNIT_FACET")) {
                        // if this is actually a block unit facet,
                        // transform all the unit lists blks on the sides into facetblks
                        UnitBlk core = null;
                        int corePosition = 0 ;
                        for (Block b : subblocks) {
                                if (b instanceof UnitBlk) {
                                        core = (UnitBlk) b;
                                        corePosition = subblocks.indexOf(b);
                                }
                        }
                        UnitFacetBlk replacement = new UnitFacetBlk(core);

                        // get the facet blocks
                        List<String> facetNames = new ArrayList<String>();
                        for (ir.Block b : subblocks) {
                                // should be a facet block
                                if (b instanceof FacetsBlk) {
                                        if (subblocks.indexOf(b) < corePosition) // left side
                                                replacement.setLeft((FacetsBlk) b);
                                        else
                                                replacement.setRight((FacetsBlk) b);// right side
                                }
                                else if (! (b instanceof UnitBlk)) {
                                        // extract all the facet names from the unitlistsblk and
produce a new facetsblk
                                        UnitListsBlk c = (UnitListsBlk) b;
                                        FacetsBlk facetBlks = c.getFacetBlks();
                                        if (subblocks.indexOf(b) < corePosition) // left side
                                                replacement.setLeft(facetBlks);
                                        else
                                                replacement.setRight(facetBlks);// right side
                                }
                        }

                        // now, construct up.
                        return replacement;
                }
                return null;
        }

        private FacetsBlk getFacetBlks() {
                for (Block b : subblocks) {
                        if (b instanceof FacetsBlk)
                                return (FacetsBlk) b;
                }
                return null;
        }

        public UnitListsBlk clone() {
                UnitListsBlk replace = new UnitListsBlk();
                for (Block b : subblocks)
                        replace.add(b.clone());

                for (CellUnit u : units)
                        replace.add(u.clone());

                replace.setType(type);

                return replace;

        }
}
```