sn*w

The language for Genetic Programming

DEVELOPED BY

Chief **sn*w** Engineer (Project Manager)	Jonathan Bell	jsb2125@columbia.edu
Staff **sn*w** Scientist (Language Guru)	Willi Ballenthin	wrb2102@columbia.edu
Principal **sn*w** Architect (Systems Design)	Vinay Bettadapura	vb2266@columbia.edu
sn*w plow (Systems Integrator)	Cesar Aguilar	caa2112@columbia.edu
Icebreaker (Tester)	Sharadh Bhaskharan	sb3053@columbia.edu

UNDER THE GUIDANCE OF

Prof. Alfred V. Aho

Lawrence Gussman Professor of Computer Science

Columbia University

Table of Contents

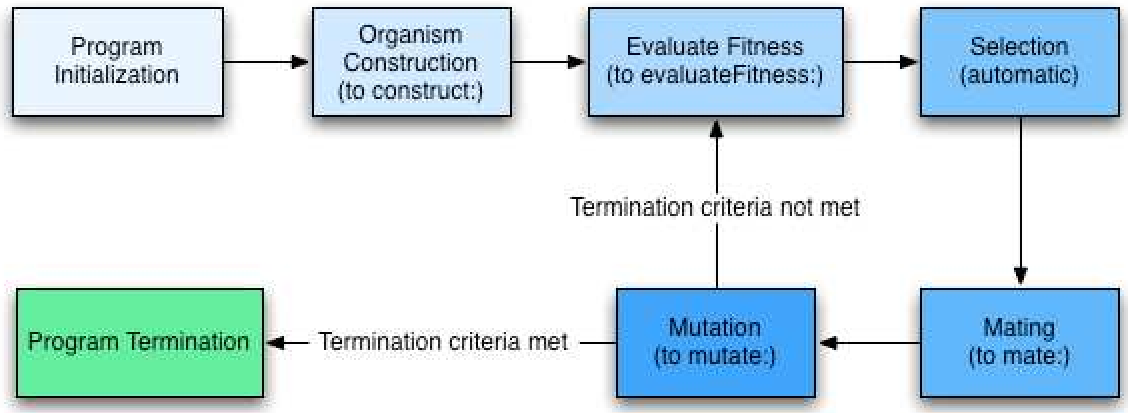
1 Introduction to sn*w

A genetic algorithm (GA) is a search algorithm in the field of artificial intelligence. GAs simulates the process of natural selection by combining elements of multiple suboptimal solutions in order to achieve better solutions. In a GA, there are a few basic key terms:

Organisms: Organisms represent possible solutions to the problem
Fitness: How well an organism meets the criteria of the problem
Chromosomes: The set of defining characteristics of an organism that makes it more or less attractive
Gene: An individual characteristic in a chromosome
Selection: The "natural death" of the unfit organisms
Mutation: The random variation in the chromosomes to maintain genetic diversity
Recombination: The process wherein two organisms are combined to create a new organism (mating)

sn*w is a declarative language that enables programmers to easily harness the power of genetic algorithms (GA). A sn*w program is a simple description of an organism structure along with simple methods for construction, mutation, selection and recombination. The sn*w compiler translates these events into a full environmental simulation.

sn*w simulations follow the following basic form:



The developer need not define each stage of the program – there are simple defaults that exist as well.

1.1 Features and Syntax Overview

The syntax of **sn*w** is heavily influenced by Lisp and Logo. It focuses on maintaining readability, with a Logo like syntax, while still holding dear to the list-based notation of Lisp. One of the key features to **sn*w** is it's built in object inference system. Genetic algorithm coding always results in performing a lot of (potentially complex) operations to every organism in a population, or every gene in a chromosome. **sn*w** leverages this repetition (or function mapping, for those familiar with Lisp) to allow for incredibly simple constructs (see the sample program at the end of this document).

1.1.1 Defining actions and types

Each **sn*w** program consists of, minimally, four actions (functions): construct, mate, mutate, getFitness. Each action is defined in a logo like syntax (see the next section, sample program for an example). A **sn*w** program must also define the structure of an organism, as well as the structure of its chromosomes.

Programmers may also extend their code by creating custom actions and additional data types, to allow for nesting. **sn*w** provides a number of built in functions (see below) as well as basic control structures (do loops, for each loops, if/then/else conditions) and recursion.

sn*w also provides debugging facilities, by providing hooks to add arbitrary output statements.

2 Language Tutorial

2.1 Hello World

This is the simplest application that can be written in **sn*w**. It simply runs and prints out "hello world". This is not very helpful, since **sn*w** provides so much more to the programmer through its event-based structure, but it is presented nonetheless:

```
#|
a simple hello world program would allow an empty simulation to run, by de-
fining nothing, and then specifying that when the machine finishes "simulat-
ing" to print out "Hello World"
|#
after termination
      print "Hello Word\n"
end
```

2.2 Accessing variables and events

At this point, we're going to take a look at a basic **sn*w** program that runs a simulation. The simulation will be very simple – it will run for 800 generations, doing nothing each time. After each generation, the program will print out the generation number.

2.2.1 Initializing Parameters

The first step is to define how many generations to run for. This is done through the global variable, `~maxGenerations`

```
~endGenerations= 800
```

To clarify that we are limiting the simulation be number of generations and not by some maximum fitness value, we clearly set `~endFitness` to -1 to imply that the simulation should NOT attempt to maximize the fitness.

```
~endFitness = -1
```

We tell **sn*w** that we want only 100 organisms in our population:

```
~populationSize = 100
```

Next, we tell **sn*w** that a chromosome to be simply two genes. Since we don't define a gene, it's assumed to be an atom.

```
define chromosome:
  2 gene
end
```

We could, if we wanted to, define an organism explicitly, or use the default provided by **sn*w**. This is the default definition:

```
define organism:
   chromosome
   name
end
```

2.2.2 Defining a Constructor

The next step is to define the initializer that will be called to construct organisms, called *construct*. For this simple example, the construct method will set the fitness of the organism to one, name the organism with an ID (~organismCount is a global variable that is monotonically increasing and provides the total number of organisms created thus far) and initializes each gene to be that same ID.

```
to construct: organism

  #fitness set to 1
  organism.fitness=1

  organism.name = ~organismCount

  #use the foreach loop to iterate through every gene quickly
  foreach gene in organism.chromosome as gene1
      gene1 = organism.name
  end
  return organism
```

```
end
```

Finally, in this example, we will specify a hook to be called after each generation. This hook will simply print out the count of the number of generations:

```
after generation:
  print: ~generationCount+"\n"
end
```

2.2.3 The complete example:

```
~endGenerations= 800
~endFitness = -1
~populationSize = 100

define chromosome:
  2 gene
end

define organism:
  chromosome
   fitness
   name
end
to construct: organism
  #fitness set to 1
  organism.fitness=1

  organism.name = ~organismCount

  #use the foreach loop to iterate through every gene quickly
  foreach gene in organism.chromosome as gene1
      gene1 = organism.name
  end
  return organism
end
each generation:
  print: ~generationCount+"\n"
end
```

2.3 Binary Bit String maximization

In this slightly more complex example, we will learn how to write a simulator to achieve a real goal: to maximize the value of the sum of all of the digits of a binary string.

To do so, the basic approach is to construct 10 random bit strings, then splice them together, favoring bit strings that already have large sums.

Let’s take a look at how this is done:

2.3.1 Initializing the parameters

The starting parameters are initialized similar to in the previous example. However, instead of limiting the number of generations, we tell **sn*w** that we are aiming to maximize the fitness, with the optimal value of 8.

```
#|
These two variables define the stopping criteria. You can specify either or
both. EndGenerationswill limit the number of generations to run for (may not
be optimal!). EndFitness will limit to some fitness stopping point. If you
provide both, then it will stop at whichever comes first.
|#
~endGenerations= -1
~endFitness     = 8
~populationSize = 10
```

Again, a chromosome is defined to be a list of 8 genes. Note that in this example, we use the longhand way to write out “8 genes”. In the previous example we said simply “2 genes,” but for the sake of thoroughness, we have provided a longhand way of expressing this as well:

```
define chromosome:
 # equivalent to "8 genes"
 (gene gene gene gene gene gene gene gene)
end
```

2.3.2 Organism Construction and Fitness Evaluation

Again, similar to the last example, we provide a simple initializer for the organism, this time setting the num value of gene to a random integer between 0 and 1, inclusive on both sides. **sn*w** knows that we want an integer because we specified both 0 and 1 as integers (versus, say, 0.0, 1.0). **sn*w** knows that we want it inclusive on the high end of the range because we used the function "randI" – where *'I'* stands for *inclusive*.

```
to construct: organism
 #Set each gene to have a random num, 0 or 1
 foreach gene in organism.chromosome as gene1
  # randI - random, inclusive
  gene1 = randI: 0, 1
 end
 return organism
end
```

We define the fitness evaluation method to simply loop over each gene in the chromosome of the organism and sum it up.

```
to evaluateFitness: organism
 var cur_fit = 0

    #Loop over each gene except for the last one
    foreach gene in organism.chromosome as gene1
     cur_fit = cur_fit + gene1
    end

 return cur_fit
end
```

2.3.3 Mating

In this example, we introduce mating. Mating occurs every generation, between organisms that were selected to mate. We also introduce the *splice* function, which splices genes from two chromosomes to return a new chromosome. Splice takes 3 arguments – how frequently genes are swapped, and then two chromosomes that contain genes. The first argument can be either a decimal number between 0 and 1, saying to splice some random percent of A with B, or an integer greater or equal to 1, signifying to take the first *n* genes of A and then the next *n* genes of B.

```
to mate: organismA, organismB, newOrganism
  newOrganism.chromsome = splice (rand: 0 100) / 100, organismA.chromosome,
organismB.chromosome
 return newOrganism
end
```

2.3.4 Output

Finally, we provide two hooks to show the progress of the simulation. Each generation we print out the average fitness of all organisms, and before finishing the simulation we print out maximum fitness, which hopefully will be 8!

```
each generation:
  print: "average fitness: " + ~averageFitness
end

before termination:
 print ~endFitness
end
```

2.3.5 Complete Example

```
#|
These two variables define the stopping criteria. You can specify either or
both. EndGenerations will limit the number of generations to run for (may not
be optimal!). EndFitness will limit to some fitness stopping point. If you
provide both, then it will stop at whichever comes first.
|#
~endGenerations= -1
~endFitness     = 8
~populationSize = 10

define chromosome:
 # equivalent to "8 genes"
 (gene gene gene gene gene gene gene gene)
end

to construct: organism
 #Set each gene to have a random num, 0 or 1
 foreach gene in organism.chromosome as gene1
  # randI - random, inclusive
  gene1 = randI: 0, 1
 end
return organism
end

to evaluateFitness: organism
 var cur_fit = 0

    #Loop over each gene except for the last one
    foreach gene in organism.chromosome as gene1
     cur_fit = cur_fit + gene1
    end

  return cur_fit
end
to mate: organismA, organismB, newOrganism
  newOrganism.chromsome = splice (rand: 0 100) / 100, organismA.chromosome,
organismB.chromosome
  return newOrganism
end
each generation:
  print: "average fitness: " + ~averageFitness
end

before termination:
 print ~endFitness
end
```

2.4 The Eight Queens Problem

2.4.1 Introduction to the Eight Queens Problem

This sample program solves the *eight queens* problem. The eight queens puzzle is a classical problem: how can you place eight queens on a board so that no two queens can attack each other. This problem can be computationally expensive, given that there are 64! / (56!*8!) possible arrangements (over 4 billion arrangements). However, using genetic algorithms, a solution can be found much quicker.

The sample program represents a possible solution as an organism with 8 genes in its chromosome. The first gene represents the first column of the board, the second gene the next row, etc. Each gene takes a value from 1-8, representing which column (from the bottom) the queen in that column is placed at. A "more fit" organism has less queens attacking each other.

2.4.2 Initialization and Declaration

Following the pattern of the previous examples, we set some basic parameters. New to this example is the parameter *topParentPool*, which defines the percentage of organisms that survive selection and *mutationRate*, which defines the rate at which organisms successfully mutate. As normal, we also define a gene and a chromosome.

```
#These two variables define the stopping criteria. You can specify either or
~endGenerations= 800
~endFitness     = 0
~topParentPool  = 0.1
~mutationRate   = (rand: 1 5)/10
~populationSize = 100

#This defines a chromosome. In this case, a chromosome is composed of 8 genes
define chromosome:
 8 gene
end
```

2.4.3 Constructor definition

We use a similar constructor to previous examples, this time setting each gene to a random value from 0 to 7.

```
#This defines an organism and how it is initialized
to construct: organism
    #Set each gene to have a random value from 0-7
    foreach gene in organism.chromosome as gene1
     gene1 = rand: 0 8
    end
 return organism
end
```

2.4.4 Fitness Evaluation

In this more complex *evaluateFitness* definition, we want to find out how many pairs of attacking queens there are in this board. The maximum is 28, but then we check each pair of queens to find any non-attacking pairs, and decrement that from the fitness.

```
to evaluateFitness: organism
  var cur_fit = 28
  var i
  var j
  var gene1
  var gene2

  for i from 0 to 7
   # a deep copy. by value
   gene1 = nth: organism.chromosome, i

   for j from i+1 to 8
    gene2 = nth: organism.chromosome, j

    #Check to see if gene1 and gene2 are in the same row
    if gene1 = gene2 and I = j then
      #do nothing
    elseif gene1=gene2 then
     cur_fit = cur_fit - 1
    #Check to see if they are along the same diagonal
    elseif ((gene1.num - gene2.num) / (i - j)) = 1
     cur_fit = cur_fit - 1
    elseif ((gene1.num - gene2.num) / (i - j)) = -1
     cur_fit = cur_fit - 1

    end
   end
  end

  return 28 - cur_fit
end
```

2.4.5 Mating and Mutation

Mating and mutation is somewhat simple. To mate, we want to get a new child that is the combination of two parents, split at some point. We use the *splice* function again for this purpose.

To mutate, we just randomly change one of the genes.

```
to mate: organismA organismB child
 #Randomly splice 2 organisms to get the child
 child.chromosome = splice: (rand: 0 8) organismA.chromosome orga-
nismB.chromosome
  return child
end

to mutate: organismA
   #mutate the organism by randomly changing a gene
   setNth: organismA.chromosome,(rand: 0 8), rand: 0 8
   return organismA
end
```

2.4.6 Output

The output from this program is just a simple print of the average fitness of the population each round, as well as a list of all of the queen locations.

```
before generation:
  print "average fitness: " + averageFitness
end

after termination:
      printPopulation:
      var solCount = 0
      foreach organism in ~population as org1
            if (evaluateFitness: org1) = 0 then
            #this is the optimal solution, print it out
            solCount = solCount + 1
            print: "Optimal Solution " + solCount +  " \n"
            print: "############################ \n"
                  var i,row,col
                        for i from 0 to 8
                        row = i
                        col = nth: org1.chromosome, i
                        print: "queen at (" + row +  " , "  + col + ")\n"
                  end
            print: "############################ \n"
            end
      end
      if solCount = 0 then
```

```
            print: "No optimal solution was found\n"
      end
end
```

2.4.7 The Complete Example

```
#These two variables define the stopping criteria. You can specify either or
~endGenerations= 800
~endFitness     = 0
#~topParentPool  = 0.1
~mutationRate   = (rand: 1 5)/10
~populationSize = 100

#This defines a chromosome. In this case, a chromosome is composed of 8 genes
define chromosome:
      8 gene
end

define organism:
      1 chromosome
end
#This defines an organism and how it is initialized
to construct: newOrganism
      #Set each gene to have a random value from 0-7
      foreach gene in newOrganism.chromosome as gene1
            gene1 = rand: 0 8
      end
      return newOrganism
end

to evaluateFitness: checkOrganism
      var curfit = 28
      var i,j,gene1,gene2

      for i from 0 to 7
            # a deep copy. by value
            gene1 = nth: checkOrganism.chromosome, i

            for j from i + 1 to 8
                  gene2 = nth: checkOrganism.chromosome, j

                  if gene1 = gene2 and i = j then
```

```
                        #do nothing
                  #Check to see if gene1 and gene2 are in the same row
                  elseif gene1 = gene2 then
                        curfit = curfit - 1
                  #check for same column!
                  #elseif i = j then
                  #     curfit = curfit - 1
                  #Check to see if they are along the same diagonal
                  elseif ((gene1 - gene2) / (i - j)) = 1 then
                        curfit = curfit - 1
                  elseif ((gene1 - gene2) / (i - j)) = -1 then
                        curfit = curfit - 1
                  end
            end
      end

      return 28 - curfit
end

to mate: organismA organismB child
      #Randomly splice 2 organisms to get the child
      child.chromosome = splice: (rand: 0 8) organismA.chromosome orga-
nismB.chromosome
      return child
end

to mutate: organismA
      #mutate the organism by randomly changing a gene
      setNth: organismA.chromosome,(rand: 0 8), rand: 0 8
      return organismA
end

before generation:
      print: "average fitness: " + ~averageFitness
end

after termination:
      printPopulation:
      var solCount = 0
      foreach organism in ~population as org1
            if (evaluateFitness: org1) = 0 then
            #this is the optimal solution, print it out
            solCount = solCount + 1
            print: "Optimal Solution " + solCount +  " \n"
            print: "############################ \n"
                  var i,row,col
                        for i from 0 to 8
                        row = i
                        col = nth: org1.chromosome, i
                        print: "queen at (" + row +  " , "  + col + ")\n"
                  end
            print: "############################ \n"
            end
      end
      if solCount = 0 then
            print: "No optimal solution was found\n"
      end

end
```

3 Language Reference Manual

3.1 Lexical Conventions

3.1.1 Comments

Single line comments are declared with a hash (#) at the beginning of the line. Block comments begin with a #| and end with a |#

Example:

```
#This is a comment
#| This is a really
Long
Comment
|#
```

3.1.2 Identifiers

Identifiers for variables and methods are case sensitive, alpha numeric, and must start with a letter. Reserved identifiers (either procedures that are required to be declared and have a special purpose, or variables that are system generated) begin with a ~ to avoid conflict.

```
IDENTIFIER: [a-zA-Z]([a-zA-Z0-9]?(\.[a-zA-Z0-9])?)*
SYS-IDENTIFIER: ~[a-zA-Z][a-zA-Z0-9]*
```

The following identifiers are provided as global names:

```
~populationSize
~generationCount
~topParentPool
~bottomParentPool
~organismLifespan
~mutationRate
~selectMethod
~maxGenerations
~maxFitness
~endGenerations
~endFitness
~generation
```

```
~mutate
~mate
~select
~organismConstructed
~organismKilled
~organismMutated
~organismMatesWithOrganism
~childBorn
~organismFitnessChanges
~termination
```

3.1.3 Keywords

The following are reserved keywords in **sn*w**:

```
to
for
each
do
end
define
var
if
else
elseif
elsif
while
return
nil
true
false
and
or
```

3.1.4 Operators

3.1.4.1 Arithmetic Operators

sn*w contains basic arithmetic operators:

+ (addition)

- (subtraction)

/ (division)

* (multiplication)

% (modulo)

^ (exponential)

++ (postfix increment)

-- (postfix decrement)

3.1.4.2 Comparison Operators

Everything in sn*w is compared by value. We provide the following comparison operators:

>

<

=

>=

<=

!=

3.1.4.3 Logical Operators

sn*w provides simple logical and, or, and not. We allow the user to choose to use simple textual and, or, and not, as well as C-like &&, ||, !. They are equivalent.

and

or

not

&&

||

!

3.1.4.4 Assignment Operator

sn*w has only one assignment operator, the equals sign. You may not use assignment within a comparison expression – we use the plain equals for comparison testing as well.

=

3.1.4.5 List operators

We provide two simple list operators, since **sn*w** is a list-based language. A << b pushes b onto the end of list a, and a >> b pops the last element off of a and onto b.

```
<< Push
>> Pop
```

3.1.5 String Literals

Strings are enclosed in either double or single quotes. “ or ‘. Escaping is normal as to C (back-slash). The following are equivalent: “this is jon’s test” and ‘this is jon\’s test’

STRING: “([^”]|(\”))*” | ‘([^’]|(\’))*’

3.1.6 Numeric Constants

Numbers are represented as either integers or floating point numbers, as a sequence of digits, possibly followed by a decimal point and then more digits.

```
NUMERIC: -?\d*\.?\d*
```

3.2 Types

sn*w has two primary types: an “atom” – which is a single item, and a “pair” – which is a pair that can contain atoms and pairs. **sn*w** also has *molecules*, which are complex combinations of atoms and pairs.

3.2.1 Atoms

An atom is the basic “untyped type” of **sn*w**. Atoms are numbers (floating point or integer), or character strings. The user need not define the type explicitly. If a user uses a quoted string, then it is assumed to be a character string. For example: “123” is treated as the string containing 123, and 123 is treated as the number one hundred twenty three.

```
ATOM:
      NUMERIC
      | STRING
```

3.2.2 Pair

A pair is the building block for the type structure of **sn*w**. Pairs can contain atoms and also more pairs. The basic idiom for constructing a list is to create a pair of (atom, *pair*), where *pair* is another pair containing another atom, and possibly another pair. List functions that are provided in **sn*w** build lists of the form (atom, (atom, (atom, (atom, nil)))), which can conveniently be written as (atom, atom, atom, atom), or (atom, atom, atom, atom, nil).

```
COMMAQ:
      COMMA
      |(epsilon)
PAIR:
      LPAREN ATOM COMMAQ NIL RPAREN
      | LPAREN ATOM COMMAQ ATOM RPAREN
      | LPAREN ATOM COMMAQ PAIR RPAREN
      | LPAREN ATOMS RPAREN
ATOMS:
      ATOM COMMAQ ATOMS
      |(epsilon)
```

3.2.3 Molecules

A molecule can be defined as follows:

```
define myMolecule:
      8 something
      10 somethingElse
      (one two three)
end
```

```
MOLECULE-DECLARATOR:
      DEFINE IDENTIFIER COLON NEWLINE MOLECULE-DEFS END
MOLECULE-DEFS:
      MOLECULE-DEFS NEWLINE
      | MOLECULE-DEF
MOLECULE-DEF:
      NUMBER IDENTIFIER
      | ATOM
```

```
        | PAIR
```

3.3 Expressions

The basic expression definition derives all expressions.

```
expression:
        assignment-expression
```

3.3.1 Primary Expressions

A primary expression is the base expression

```
primary-expression:
        ATOM
        | PAIR
        | IDENTIFIER
        | LPAREN EXPRESSION RPAREN
```

3.3.2 Postfix Expressions

Postfix expressions have the next lowest level of precedence

```
postfix-expression:
        primary-expression
        | postfix-expression PLUSPLUS
        | postfix-expression MINUSMINUS
        | function-expression
```

3.3.3 Function Calls

Function calls have the same precedence as postfix expressions, and take the form below:

An example function call:

var x = doSomethingTo: organism, 5, 6

Note that the method name is doSomethingTo, and there are 3 arguments: organism, 5, and 6. It would be equivalent to write:

```
var x = doSomethingTo: organism 5 6
```

```
COMMAQ:
      ,
      | (epsilon)
function-expression:
      function-name COLON PARAMS
PARAMS:
      PARAM COMMAQ PARAMS
      | (epsilon)
PARAM:
      ATOM
      | PAIR
      | IDENTIFIER
```

3.3.4 Unary Operator Expressions

Unary expressions are simple logical NOT, or arithmetic negation expressions.

```
unary-expression:
      postfix-expression
      | MINUS unary-expression
      | NOT unary-expression
```

3.3.5 Binary Operator Expressions

Binary expressions, in order of increasing precedence, are the arithmetic operators +, -, /, *, %, and ^.

```
binary-expression:
      additive-expression
additive-expression:
multiplicative-expression
      | additive-expression + multiplicative-expression
      | additive-expression - multiplicative-expression
multiplicative-expression:
      multiplicative-expression * pow-expression
      | multiplicative-expression / pow-expression
      | multiplicative-expression % pow-expression
      | pow-expression
pow-expression:
      unary-expression
      | pow-expression ^ unary-expression
```

3.3.6 Relational Operator Expressions

Relational expressions encompass all comparisons.

```
relational-expression:
      additive-expression
      | relational-expression < additive-expression
      | relational-expression > additive-expression
      | relational-expression <= additive-expression
      | relational-expression >= additive-expression
equality-expression:
      relational-expression
      | equality-expression = relational-expression
      | equality-expression != relational-expression
```

3.3.7 Logical Operator Expressions

Logical expressions allow expressions to be joined by logical AND and OR

```
logical-and-expression:
      equality-expression
      | logical-and-expression AND equality-expression
logical-or-expression:
      logical-and-expression
      | logical-or-expression OR logical-and-expression
```

3.3.8 Assignment expressions

Assignment expressions are the highest binding precedence.

```
assignment-expression:
      logical-or-expression
      | IDENTIFIER = assignment-expression
```

3.4 Declarations

3.4.1 Actions (functions)

Actions are declared with the keyword to. Parameters are specified after the colon, comma separated.

```
to doSomething: org1, org2
      …
end
```

```
declarator:
      TO IDENTIFIER COLON PARAMS NEWLINE STATEMENTS END
```

3.4.2 Variables

Variables are declared with the var keyword. Variables are either atoms or pairs.

```
var-declarator:
      var IDENTIFIER
      | var assignment-expression
```

3.4.3 Pairs

Pairs can be declared as follows:

```
COMMAQ:
      COMMA
      |(epsilon)
PAIR:
      LPAREN ATOM COMMAQ NIL RPAREN
      | LPAREN ATOM COMMAQ ATOM RPAREN
      | LPAREN ATOM COMMAQ PAIR RPAREN
      | LPAREN ATOMS RPAREN
ATOMS:
      ATOM COMMAQ ATOMS
      |(epsilon)
```

3.4.4 Molecules

Molecules in **sn*w** are combinations of atoms. They can be defined as follows:

```
MOLECULE-DECLARATOR:
      DEFINE IDENTIFIER COLON NEWLINE MOLECULE-DEFS END
MOLECULE-DEFS:
      MOLECULE-DEFS NEWLINE
      | MOLECULE-DEF
MOLECULE-DEF:
      NUMBER IDENTIFIER
      | ATOM
      | PAIR
```

3.4.5 Debug hooks

Debug hooks are defined in the form:

[before | after] each *event*, where *event* is one of the predefined events:

```
EVENTNAME:
      generation
      | mutation
      | mating
      | selection
      | organismConstructed
      | organismKilled
      | organismMutated
      | organismMatesWithOrganism
      | childBorn
      | organismFitnessChanges
debug-hook-declare:
      TIME-SEQ each EVENTNAME COLON STATEMENTS END
TIME-SEQ:
      BEFORE
      | AFTER
```

3.5 Statements

A statement is the basic building block of the program, combining control structure and expressions.

```
statement:
      expression-statement
      | selection-statement
      | iteration-statement
statements:
      statement NEWLINE statements
      | statement
```

3.5.1 Expression Statements

```
expression-statement:
      expression
```

3.5.2 Selection Statements

Selection statements are simple if statements. **sn*w** if statements do not require parentheses. Instead, the condition statement is terminated by the keyword “then”. Elseif statements can be written as “else if”, “elseif”, or “elsif”.

```
selection-statement:
      IF partial-selection-statement END
partial-selection-statement:
      logical-or-expression THEN STATEMENTS
      | logical-or-expression THEN STATEMENTS ELSE STATEMENTS
      | logical-or-expression THEN STATEMENTS ELSEIF partial-selection-
statement
ELSEIF:
      elseif
      | elsif
```

3.5.3 Iteration Statements

sn*w provides 3 basic loop structures: while, for, and foreach. While is a traditional while loop (no parentheses necessary, but requiring a new line after the condition). For is a simple construct that allows you to increment a variable from some expression to another expression, and then use that variable inside of your loop. Foreach is a handy construct that allows the user to iterate over each x of y, optionally specifying boundaries.

```
iteration-statement:
      while logical-or-expression NEWLINE STATEMENTS END
      | for IDENTIFIER from expression to expression by-statement NEWLINE
STATEMENTS END
      | foreach IDENTIFIER in-statement as-statement from-statement NEWLINE
STATEMENTS END
as-statement:
      as IDENTIFIER
      | (epsilon)
in-statement:
      in IDENTIFIER
      | (epsilon)
from-statement:
      from expression to expression
      | (epsilon)
by-statement:
      by expression
```

```
        | (epsilon)
```

3.6 Scoping

3.6.1 General Scoping

Scope in **sn*w** is very simple: all variables declared outside of a method are global. All variables declared within a method are local to the method. If you declare a variable within a method with the same name as a file-scoped variable, it will hide the file-scoped variable.

3.6.2 "Global" Scope

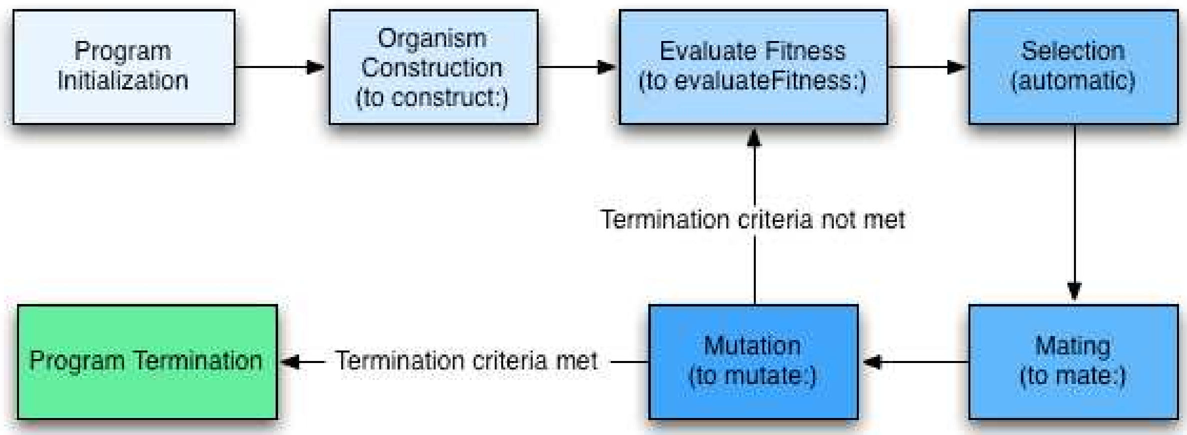
Some variables come free with **sn*w**, and are provided to the user as regular variables which begin with a ~. These variables are always accessible and cannot be hidden.

3.6.3 Block Scope

Within a block – that is, an iteration or selection statement - variables that are declared are local to the block.

3.7 Program Structure

A **sn*w** program is event based, and is formed by defining event responders and event criteria. We have provided a simple block diagram that shows the flow between events.



3.8 Grammar

The complete grammar for **sn*w**, as defined above, follows.

3.8.1 Lex Tokenizer

```
package com.google.code.pltsnow.gen;
%%

%byaccj

%{
//Authored by Jon, Cesar, Vinay, Sharadh
  /* store a reference to the parser object */
  private static Parser yyparser;
  public int ln = 0;
  /* constructor taking an additional parser object */
  public Yylex(java.io.Reader r, Parser yyparser) {
    this(r);
    this.yyparser = yyparser;
  }
%}

digit=            [0-9]
ncc=        [^#|]|\n
%%

\#.*\n                                    {ln++; yyparser.yylval = new Par-
serVal(""); return yyparser.NEWLINE;}
#\|(#|{ncc})*\|\|*({ncc}(#|{ncc})*\|\|*)*#      {ln++; yyparser.yylval = new
ParserVal(""); return yyparser.NEWLINE;}
to                              {     return yyparser.TO;}
for                       {     return yyparser.FOR;}
foreach                         {     return yyparser.FOREACH;}
each                      {     return yyparser.EACH;}
while                     {     return yyparser.WHILE;}
return                          {     return yyparser.RETURN;}
do                        {     return yyparser.DO;}
end                       {     return yyparser.END;}
define                          {     return yyparser.DEFINE;}
var                       {     return yyparser.VAR;}
if                        {     return yyparser.IF;}
then                      {     return yyparser.THEN;}
else                      {     return yyparser.ELSE;}
elseif                          {     return yyparser.ELSIF;}
elsif                     {     return yyparser.ELSIF;}
```

```
from                    {      return yyparser.FROM;}
as                      {      return yyparser.AS;}
in                      {      return yyparser.IN;}
by                      {      return yyparser.BY;}
nil                     {      return yyparser.NIL;}
true                    {      return yyparser.TRUE;}
false                   {      return yyparser.FALSE;}
before                        {      return yyparser.BEFORE;}
after                   {      return yyparser.AFTER;}
"++"                          {      return yyparser.PLUSPLUS;}
"--"                          {      return yyparser.MINUSMINUS;}
"+"                           {      return yyparser.PLUS;}
"-"                           {      return yyparser.MINUS;}
"*"                           {      return yyparser.MUL;}
"/"                           {      return yyparser.DIV;}
"%"                           {      return yyparser.MODULO;}
"^"                           {      return yyparser.POWER;}
"<<"                          {      return yyparser.LIST_OP_PUSH;}
">>"                          {      return yyparser.LIST_OP_POP;}
"<="                          {      return yyparser.REL_OP_LE;}
"<"                           {      return yyparser.REL_OP_LT;}
">="                          {      return yyparser.REL_OP_GE;}
">"                           {      return yyparser.REL_OP_GT;}
"="                           {      return yyparser.EQUALS;}
"!="                          {      return yyparser.REL_OP_NE;}
"and"|"&&"                    {      return yyparser.LOG_OP_AND;}
"or"|"||"                           {      return yyparser.LOG_OP_OR;}
"not"|"!"                           {      return yyparser.LOG_OP_NOT;}
"("                           {      return yyparser.LPAREN;}
")"                           {      return yyparser.RPAREN;}
":"                           {      return yyparser.COLON;}
","                           {      return yyparser.COMMA;}
\.                            {      return yyparser.DOT;}
genera-
tion|termination|mutation|mating|selection|organismConstructed|organismKilled
|organismMutated|organismMatesWithOrganism|childBorn|organismFitnessChanges
      {      yyparser.yylval = new ParserVal(yytext()); return yypars-
er.EVENT_NAME_IDENTIFIER;}
[a-zA-Z][a-zA-Z0-9]*                {      yyparser.yylval = new Parser-
Val(yytext()); return yyparser.IDENTIFIER;}
\~[a-zA-Z][a-zA-Z0-9]*              {      yyparser.yylval = new Parser-
Val(yytext()); return yyparser.SYS_IDENTIFIER;}
```

```
\"([^\"]|\\\"|\\n)*\"|\'([^\']|\\\'\\n)*\'      {     yyparser.yylval = new
ParserVal(yytext()); return yyparser.STRING;}
-?{digit}*(\.{digit}+)?             {     yyparser.yylval = new Parser-
Val(yytext()); return yyparser.NUMERIC;}
\n                            {     ln++; yyparser.yylval= new Parser-
Val("\n"); return yyparser.NEWLINE;}
[ \t]                         {}
.                             { return 0;}
```

3.8.2 Yacc Grammar

```
%{
import java.lang.Math;
import java.io.*;
import java.util.StringTokenizer;
import java.util.Scanner;
//Authored by Jon, Cesar, Vinay, Sharadh
%}



%token TO FOR FOREACH WHILE RETURN DO END DEFINE VAR IF THEN ELSE ELSIF FROM
AS BY BEFORE AFTER EACH IN
%token PLUSPLUS MINUSMINUS PLUS MINUS MUL DIV MODULO POWER LIST_OP_PUSH
LIST_OP_POP REL_OP_LE REL_OP_GE REL_OP_LT REL_OP_GT REL_OP_NE EQUALS
LOG_OP_AND LOG_OP_OR LOG_OP_NOT LPAREN RPAREN COLON COMMA DOT
%token TRUE FALSE NIL
%token NEWLINE SPACE
%token EVENT_NAME_IDENTIFIER
%token RESERVED_IDENTIFIER
%token IDENTIFIER
%token SYS_IDENTIFIER
%token STRING
%token NUMERIC

%%
program_program : program_unit {
System.out.println("import com.google.code.pltsnow.snowfield.*;");
System.out.println("import com.google.code.pltsnow.gen.*;");
System.out.println("import java.util.ArrayList;");

System.out.println("public class SnowProgramImp extends BaseSnowProgram
{\n");
System.out.println($1.sval);
System.out.println("public static void main(String[] args)
{\n\tSnowProgramImp this_prog = new SnowProgramImp();
\n\tthis_prog.startProgram();\n}");
System.out.println("\n}");
}
program_unit            :     external_declaration {$$=$1;}
                  |     program_unit external_declaration {$$ = new Parser-
Val($1.sval + $2.sval);}
                  ;

external_declaration    :
                        function_declarator {$$=$1;}
                  |     var_declarator {$$=$1;}
                  |     molecule_declarator {$$=$1;}
                  |     debug_hook_declarator {$$=$1;}
                  |     global_variable_assignment {$$=$1;}
                  |     NEWLINE {$$=$1;}
                  |       END {}
                  |     error NEWLINE { System.err.println("Line Number:" +
lexer.ln + "\t Error:Error in External Definition");}
                  ;

global_variable_assignment    :
                        SYS_IDENTIFIER EQUALS expression { $$ = assignVaria-
ble($1,$3); }
                  ;

atom              :
                        NUMERIC {$$.sval= "new SnowAtom("+$1.sval+")";}
                  |     STRING {$$.sval= "new SnowAtom("+$1.sval+")";}
                  |     IDENTIFIER {$$.sval = $1.sval;}
                  ;

commaq                  :
                              COMMA {$$= new ParserVal(",");}
                  |           {$$=new ParserVal(",");}
                  ;

pair              :
                        LPAREN atom commaq NIL RPAREN   { $$.sval = $2.sval;}
                  |     LPAREN atom commaq atom RPAREN  { $$.sval = $2.sval +
"," + $4.sval;}
```

```
                  |     LPAREN atom commaq pair RPAREN  { $$.sval = $2.sval +
"," + $4.sval;}
                  |     LPAREN atoms RPAREN     { $$.sval = $2.sval;}
                  ;

atoms             :
                        atom commaq atoms { $$.sval = $1.sval + "," +
$3.sval;}
                  |     atom              { $$.sval = $1.sval;}
                  ;



primary_expression      :
                        atom {$$=$1;}
                  |     pair {$$=$1;}
                  |     identifier {$$=$1;}
                  |     LPAREN expression RPAREN {$$.sval = "(" + $2.sval +
")";}
                  ;

identifier        :
                        compound_identifier {$$= $1;}
                  |     SYS_IDENTIFIER { $$ = new Parser-
Val("symbols.get(\""+$1.sval+"\")");}
                  ;

compound_identifier     :
                        compound_identifier DOT IDENTIFIER { $$ = buildCom-
poundIdentifier($1,$3); }
                  |     IDENTIFIER {$$ = $1;}
                  ;
postfix_expression      :
                        primary_expression {$$=$1;}
                  |     postfix_expression PLUSPLUS {$$ =
doOp("increment",$1,null); }
                  |     postfix_expression MINUSMINUS {$$ =
doOp("decrement",$1,null); }
                  |     function_expression {$$=$1;}
                  ;

function_expression     :
                        IDENTIFIER COLON params_opt { $$ = executeFunc-
tion($1,$3); }
                  ;
```

```
params_opt          :     params {$$ = $1;}
                    |           {$$.sval = "";}
                    ;

params                    :
                          params commaq param {$$ = new Parser-
Val($1.sval+","+$3.sval);}
                    |     param {$$ = $1;}
                    ;

param               :
                          expression {$$ = $1; }
                    ;

unary_expression    :
                          postfix_expression {$$=$1;}
                    |     MINUS unary_expression {$$ = doOp("minus",new Parser-
Val("0"),$2);}
                    |     LOG_OP_NOT unary_expression {$$ =
doOp("log_not",$2,null);}
                    ;



additive_expression       :
                          multiplicative_expression {$$=$1;}
                    |     additive_expression PLUS multiplicative_expression
{$$ = doOp("plus",$1,$3);}
                    |     additive_expression MINUS multiplicative_expression
{$$ = doOp("minus",$1,$3);}
                    ;

multiplicative_expression:
                          multiplicative_expression MUL pow_expression {$$ =
doOp("times",$1,$3);}
                    |     multiplicative_expression DIV pow_expression {$$ =
doOp("divide",$1,$3);}
                    |     multiplicative_expression MODULO pow_expression {$$ =
doOp("modulo",$1,$3);}
                    |     pow_expression {$$=$1;}
                    ;
pow_expression            :
                          unary_expression {$$=$1;}
```

```
                  |     pow_expression POWER unary_expression {$$ =
doOp("power",$1,$3);}
                  ;

relational_expression   :
                        additive_expression {$$ = $1;}
                  |     relational_expression REL_OP_LT additive_expression
{$$ = doOp("lt",$1,$3);}
                  |     relational_expression REL_OP_GT additive_expression
{$$ = doOp("gt",$1,$3);}
                  |     relational_expression REL_OP_LE additive_expression
{$$ = doOp("le",$1,$3);}
                  |     relational_expression REL_OP_GE additive_expression
{$$ = doOp("ge",$1,$3);}
                  ;

equality_expression     :
                        relational_expression {$$ = $1;}
                  |     equality_expression EQUALS relational_expression {$$
= doOp("equals",$1,$3);}
                  |     equality_expression REL_OP_NE relational_expression
{$$ = doOp("nequals",$1,$3);}
                  ;

logical_and_expression:
                        equality_expression {$$=$1;}
                  |     logical_and_expression LOG_OP_AND equality_expression
{$$ = doOp("log_and",$1,$3);}
                  ;

logical_or_expression   :
                        logical_and_expression {$$=$1;}
                  |     logical_or_expression LOG_OP_OR logi-
cal_and_expression {$$ = doOp("log_or",$1,$3);}
                  ;

assignment_expression   :
                        compound_identifier EQUALS expression { $$ = assign-
Variable($1,$3); }
                  |     compound_identifier LIST_OP_PUSH expression { $$ =
Push($1,$3); }
                  |     compound_identifier LIST_OP_POP compound_identifier {
$$ = Pop($1,$3); }
                  ;
```

```
statement          :
                        expression_statement NEWLINE {$$=addLineEnding($1);}
                   |    selection_statement NEWLINE {$$=$1;}
                   |    iteration_statement NEWLINE {$$=$1;}
                   |    var_declarator NEWLINE {$$=addLineEnding($1);}
                   |    assignment_statement NEWLINE {$$=addLineEnding($1);}
                   |    return_statement NEWLINE {$$=$1;}
                   |    NEWLINE
                   |    error NEWLINE { System.err.println("Line Number:" +
lexer.ln + "\t Error:In Statement");}
                   ;
return_statement:
                              RETURN expression {$$.sval = "return " +
$2.sval + ";\n";}
                   ;
statements         :
                        statement { $$=$1; }
                   |    statements statement {$$ =new Parser-
Val($1.sval+$2.sval);}
                   ;

assignment_statement    :
                        assignment_expression {$$=$1;}
                   ;

expression_statement    :
                        expression {$$=$1;}
                   ;



expression         :
                        logical_or_expression {$$=$1;}
                   ;

selection_statement     :
                        IF partial_selection_statement END { $$ = makeFul-
lIfStatement($2); }
                   ;

partial_selection_statement:
                        expression THEN statements {$$ = makePartialI-
fElse($1,$3, null);}
```

```
                  |     expression THEN statements ELSE statements {$$ = ma-
kePartialIfElse($1,$3, $5);}
                  |     expression THEN statements ELSIF par-
tial_selection_statement {$$ = makePartialIfElseIf($1,$3, $5);}
                  ;

iteration_statement     :
                        WHILE expression NEWLINE statements END { $$ = doW-
hile($2,$4); }
                  |     FOR identifier FROM expression TO expression
by_statement NEWLINE statements END { $$ = doFor($2,$4,$6,$7,$9); }
                  |     FOREACH identifier in_statement as_statement
from_statement NEWLINE statements END { $$ = doForeach($2,$3,$4,$5,$7); }
                  ;

as_statement            :
                        AS identifier     { $$ = $2; }
                  |                 {$$ = null; }
                  ;

in_statement            :
                        IN identifier { $$ = $2; }
                  | {$$ = null; }
                  ;

from_statement          :
                        FROM expression TO expression { $$ = new Parser-
Val($2.sval + ":" + $4.sval); }
                  | {$$ = null; }
                  ;

by_statement            :
                        BY expression     { $$ = $2;}
                  |     { $$ = null;}
                  ;

debug_hook_declarator   :
                        time_seq EVENT_NAME_IDENTIFIER COLON NEWLINE state-
ments END {
                        $$ = createDebugHook($1,$2,$5);
                        }
                  ;
```

```
time_seq          :
                        BEFORE {$$ = new ParserVal("before"); }
                  |     AFTER {$$ = new ParserVal("after"); }
                  ;

function_declarator     :
                        TO IDENTIFIER COLON params NEWLINE statements END {
$$ = createFunction($2,$4,$6); }
                  ;
var_declarator          :
                        VAR declaration_list {$$ = $2;}
                  ;

declaration_list  :
                        declarator {$$ = $1;}
                  |     declaration_list COMMA declarator {$$.sval = $1.sval
+$3.sval;}
                  ;

declarator        :
                        IDENTIFIER { $$ = declareLocalVariable($1); }
                  |     IDENTIFIER EQUALS expression { $$ = declareLocalVari-
able($1,$3); }
                  ;

molecule_declarator     :
                        DEFINE IDENTIFIER COLON NEWLINE molecule_defs END {
$$ = defineMolecule($2,$5); }
                  ;

molecule_defs           :
                        molecule_def {$$ = $1;}
                  |     molecule_defs molecule_def { $$= new Parser-
Val($1.sval +$2.sval); }
                  ;

molecule_def            :
                        NUMERIC IDENTIFIER NEWLINE { $$ = moleLazy-
Create($1,$2);}
                  |     identifier NEWLINE { $$ = moleCreateOne($1);  }
                  |     pair NEWLINE { $$ = moleCreateFromPair($1);  }
                  |     NEWLINE
                  ;
```

```
%%

      protected Parser(Reader r) {
          lexer = new Yylex(r, this);
        }
      public static void main(String args[]) throws IOException {
          Parser yyparser = new Parser(new FileReader(args[0]));
          yyparser.yyparse();
        }
```

4 Project Plan

4.1 Process

From the outset, we decided to follow an agile software process. We began meeting early – right after we formed our team to begin brainstorming languages. We each proposed a language concept, met as a group, narrowed it down to two languages, then eventually down to just **sn*w**.

We met weekly in my apartment, where I have a large projection screen, whiteboard, and ample seating. In between meetings, we created a "punch list" of tasks to get taken care of for the next time that we came together. At the beginning this meant thinking about language constructs and towards the end this meant implementing specific constructs or identifying and fixing bugs. We took full advantage of our meeting time (which generally ranged from one to five hours on Sunday evening) to work on architectural design issues as a group. For example, we wrote the entire grammar in one sitting, all together, but split up the translation rules between team members.

4.2 Responsibilities

Our official team breakdown is:

Jonathan Bell – Project Manager

Willi Ballenthin – Language Guru

Vinay Bettadapura – Systems Architect

Cesar Aguilar – Systems Integrator

Sharadh Bhaskharan – Tester

In reality, most of these roles were completely blurred. Sharadh and Vinay implemented the majority of the translation rules, while Jonathan and Willi implemented the majority of the runtime environment. Cesar supported both sides of the operation, involving himself equally on both parts. Sharadh authored the entire testing suite and was responsible for running it. Jonathan also created graphics and coordinated all meetings, deadlines, and submissions.

4.3 Implementation Style Sheet

We formatted our code using the style-sheet built into Eclipse. It's a fairly simple, readable, and easy to follow. Here is an example method, formatted:

```
@Override
	public SnowType decrement() throws UnsupportedOperationException {
		if (isNumeric()) {
			data = minus(new SnowAtom(1)).get();
			return this;
		}

		throw new UnsupportedOperationException();
	}
```

4.4 Timeline

The following timeline shows major milestones:

1/27/09	Team Formed
1/28/09	Name chosen (snow)
2/11/09	Language scope chosen (Genetic Algorithms)
2/22/09	First Sample Program Written
2/25/09	Whitepaper Submitted
3/30/09	LRM & Tutorial Submitted
4/12/09	Target language switched to Java
4/20/09	Project presented to class
5/10/09	Language Completed!
5/11/09	Final Presentation

5 Language Evolution

The SNOW group is excited note that the initial motivations for the SNOW language remain driving force behind its current implementation. When we set out working on the design, it was envisioned that the language that might serve a useful introduction to genetic algorithms, yet also be of use to an experience researcher. Although genetic algorithms emulate quite a complex process, we saw that significant steps could be reduced to standard, but override-able, implementations. This approach bore much fruit, as changes to the language were not to its core features, but rather to the ways unique aspects of the language were implemented. On the whole, changes to the language specification seem especially relevant in light of our mid-project shift in target languages from C to Java. It came about in response to growing concerns over user defined data structures and memory management, but reflected the way in which the group valued the initial vision.

When applied to a specific but suitable search domain, genetic algorithms can effective make use of great computing resources. For this reason SNOW was to be implemented for (and in) the C language; C is often used as the standard for a high performance general language. Unfortunately, C remains infamous for its memory model, and deservingly so, given the ease at which a programmer can introduce leaks. To account for the issue of memory management in the SNOW compiler, this team switched to the garbage-collecting Java language. Java is a slower language, however its object orientation allowed for a much more faithful implementation of the language initially described by SNOW.

The transition to Java represented a refocusing from the realm of high performance and optimized code generation (of which no member had any experience) to a pattern of general purpose genetic algorithms. For example, the initial description of SNOW covered the ability of a programmer to define his own data structures. In C, this was not a possibility. However, we felt interesting data structures were more important than high performance code. Specifically, it opened the door to experimentation with genetic algorithms to a greater audience beyond academic researchers. Also, SNOW's natural constructs for the definition of structures nicely paralleled the readable syntax of SNOW. In the end, it produced a really cool type system. Structures are initialized lazily, and objects are not strictly defined. For example, setting a field of an object that doesn't explicitly have the field does not produce an error, but dynamically redefines the type.

The decision to change from C to Java was made in a similar manner as all language specification changes were. Most important was ensuring that each member of the group understood the current state of the project and the issue at hand. Early on, when not all members had a working knowledge of genetic algorithms, it was especially important to constantly reevaluate all updates in light of the big picture. Often an introductory email was sent to the SNOW mailing list, detailing the motivation for the patch, and a possible implementation. At the following team meeting, members were encouraged to present their arguments for or against the update, and illustrate how it would affect the end programmer. The resulting discussions were not only lively but served to reinforce the goals of the SNOW team. The target language debate took place over a number of meetings (and weeks), and was agreed upon only after implementations of the basic parsers were provided by each side.

Once a patch had been suggested and accepted, a team member or two would get to work implementing and documenting the change. A wiki-like webpage detailed up to date information regarding all changes to the Language Reference Manual. This was used in conjunction with Subversion logs track the progress of agreed-upon changes, and reconsider the its effects on the project as a whole. Following the successful coding and documentation of a change, a follow up email would circulate containing specific notes and considerations.

In the end, the major deciding question that each team member had to ask was, “what would the SNOW programmer do?” It was important to keep in mind the experience of the end user, and if it fit with the vision of SNOW forged during heated debate. In the end the SNOW team produced a compiler extremely similar to that described by early documents, likely due to the cooperative nature of the entire process.

6 Translator Architecture

This chapter introduces and explains the architecture of the **sn*w** translator. The architectural block diagram of the translator and the interfaces between each of the modules is explained.

6.1 Architectural Block Diagram and Interfaces

The block diagram of the **sn*w** translator along with the interfaces between each of the modules is as shown in Figure 6.1.

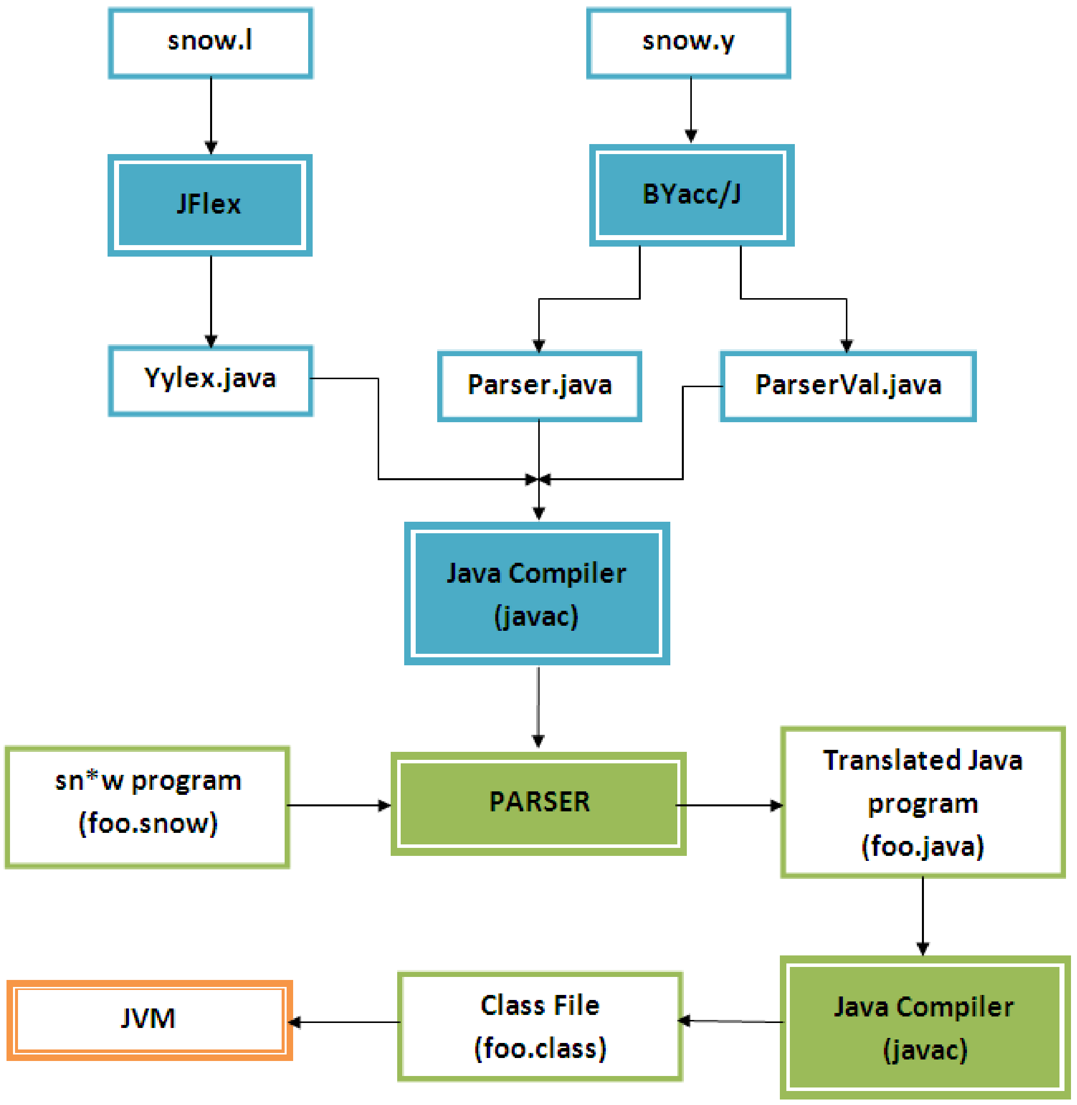


Figure 6.1: Block diagram of the **sn*w** translator.

The parser is generated using JFlex and BYacc/J. JFlex is a lexical analyzer generator for Java written in Java whereas BYacc/J is an extension of the Berkeley v1.8 YACC compatible parser generator. The lex file 'snow.l' is given as an input to JFlex which outputs Yylex.java. Similarly the file 'snow.y' is given as input to the BYacc/J program that outputs Parser.java and Parser-Val.java. When compiled together using 'javac', the Parser is generated. This generated parser can translate the **sn*w** programs into Java programs.

The generated Java programs have the following structure:

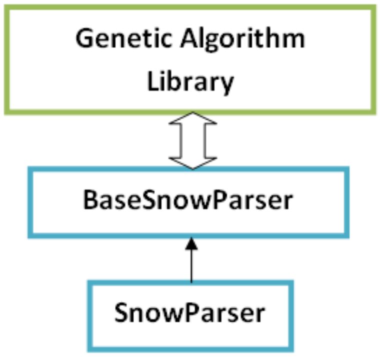


Figure 6.2: Structure of the generated Java program

The 'SnowParser' is the program that is generated. It derives from 'BaseSnowParser'. BaseSnowParser is the main class that we have written that contains all the functions that are required to compute and run the Genetic Algorithms.

6.2 Who did what?

Sharadh and Vinay wrote the initial versions of the Lexer and Parser….then Jon, Vinay, Sharadh and Ceasar sat together in Jon Bell's apartment, lovingly dubbed "Bell Labs," and changed it from C to Java. Then over a course of the next 2 to 3 weeks all 5 of us together added the semantic rules and finalized the grammar. After that Willi, Ceaser and Jon took care of the run time environment side while Sharadh and Vinay took care of the testing framework and other stuff.

7 Development Environment

In the development phases of our language we had considered multiple target languages. Some of the target languages included C, Java, and Perl, as these were languages at least a few of use knew. In the end we choose C as there was about an even split between C and Java programmers, with the belief that C would allow us to optimize easier to produces faster **sn*w** code. This was our target language until we sat down to implement our Syntax Directed Translations. While writing out our SDT in yacc we started running into memory allocation issues with variables being passed around. As our main C programmer was not around we decided to switch to Java at this point as to not deal with the details. At this point we had already written the lexical analyzer using Lex, and the grammer with yacc so we were looking for a way of exporting them into Java with minimal effort. So our main development tools at this point included JFlex which uses all the same syntax as Lex, and Byacc/J which uses much the same syntax of yacc after some minor modifications.

After this a SVN repository was setup with Google Code. We choose Google Code because it includes many nifty features that allowed us to better communicate with one another. Some of the features it included were wiki pages which allowed us to post documentation on the code thus far and how much was left. Another feature we used was email alerts so whenever one of use commit new code into the svn we would all be alerted, thus we knew what not to work on.

When we were using C there was no set development environment, at that point we allowed everyone to use there own development environment. Once we moved on to Java we decided to standardize the development environment since Java is a multi-system programming languages many of the same development tools exist for each system, unlike C. This had been an issue earlier as we were mostly using Linux/Mac machines something which changed to Linux/Mac/Windows after moving to Java. So we setup Eclipse as our programming environment, we installed the Subclipse plugin which allows us to use SVN in eclipse itself with no need of going to the command line.

As for building our project we create Makefile that would check operating system because there were different versions of Byacc/J depending on operating systems. We also included a make command to compile each sample snow program and a run to test the sample snow program once compiled. We also introduced a make clean as we need to get rid of all the class files before compiling because we noticed that SVN included our class files whenever someone added them and thus our newly compiled parser would occasionally not be the new one thus we need a quick way to get rid of all the old ones.

In summary, our development environment consisted of Eclipse with Subclipse installed for SVN support, JFLex and Byacc/J for Lexical analysis and Syntax directed translations and the Make-file to compile everything and run our test programs. Lastly we used Java 6.0, as we used some methods not found in other versions of Java.

8 Test Plan

The sn*w **test plan** highlights upon the testing approach, testing methodology, test framework and strategies used to improve testing.

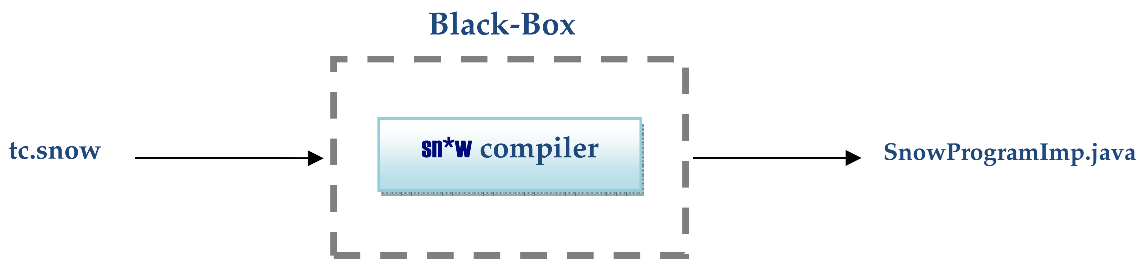
8.1.1 Approach to sn*w testing

The testing of the sn*w compiler is focused on ensuring that the following important facets of the compiler hold.

*'Has the sn*w compiler translated code correctly into the target Java program according to the semantic rules defined?'*	***'Is the generated target Java program semantically equivalent to corresponding sn*w program?'***	***'Are semantic rules associated with each production of the grammar evaluated at least once?'***

8.1.2 Testing Methodology

sn*w has been tested using black box testing methodology. It is as shown in the figure below

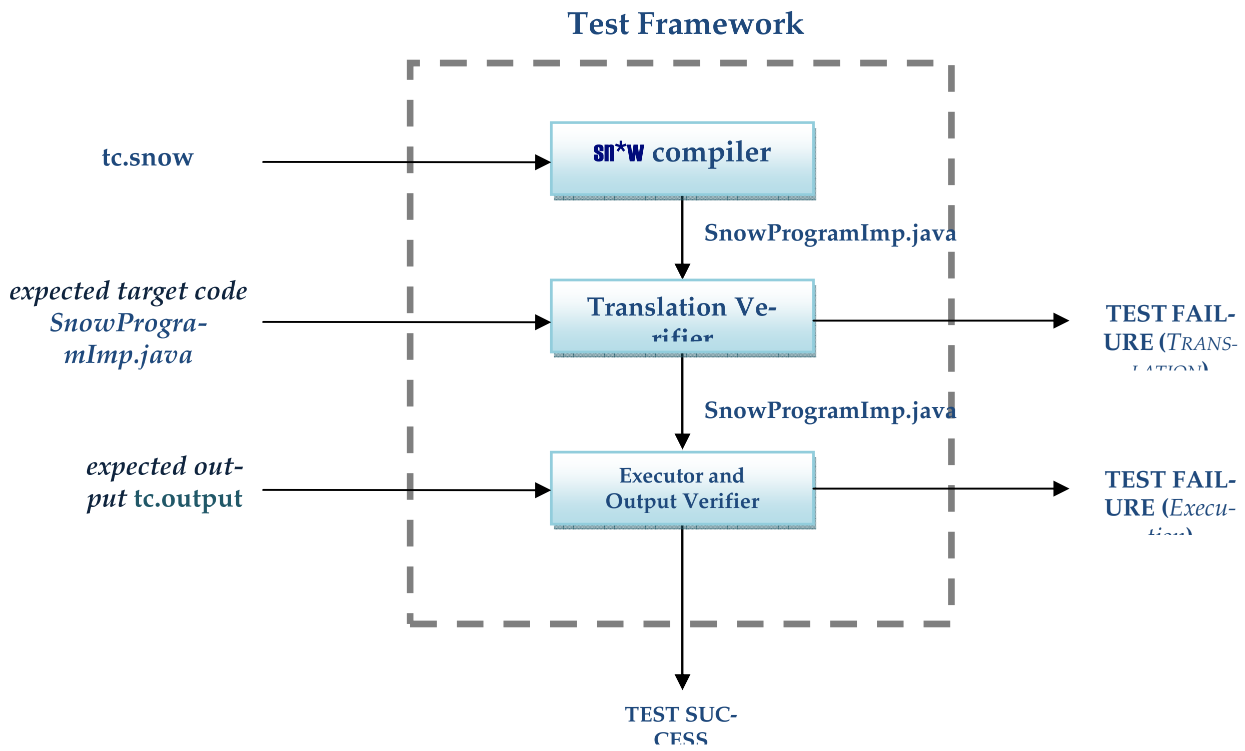


The components of each black-box test case are shown in the table below.

tc.snow	**The program written in sn*w language.**
SnowProgramImp.java	The expected target program in Java.
tc.output	The expected output on execution.

8.1.3 Test Framework

The test framework used for sn*w testing consisted of a translation verifier, an executor and output verifier. It is implemented using a simple script which uses UNIX '*diff*' command to compare expected and actual output. The output of '*diff*' command is captured in a temporary file and is available for debugging.



8.1.4 Strategies used to improve Testing

8.1.4.1 Incremental Testing

The testing of **sn*w** compiler was performed in parallel with the development. As the **sn*w** compiler went through different stages of development from the lexer, parser and SDT, the test suited was enhanced to cover test scenarios arising from the different parts of the compiler. In addition to this, the core sample programs specified in the tutorial were used to test the back-end Genetic Algorithm as it was developed.

8.1.4.2 Automated Regression Testing

To facilitate easy testing of the **sn*w** compiler a regression suite was developed. The test cases were picked to ensure maximum coverage of productions of the grammar. The regression testing was automated by creating the script **runSuite.sh** which executed, verified and reported the test results.

8.1.4.3 Multiple Perspectives

The core sample programs specified in the language tutorial were written by different members of the team. This greatly assisted testing as each team-member used various constructs of **sn*w** differently.

Test Script – runSuite.sh

```
cd ../src
test_output_file=/tmp/test.out
filename=${test_output_file}
if [ -f $filename ]
then
        rm $filename
fi

for i in sample1 sample2 sample3 sample4 tc_1 tc_2 tc_3 tc_4 tc_5 tc_6 tc_7
tc_8
do
        filename=/tmp/$i.diff
        if [ -f $filename ]
        then
                rm $filename
        fi
done

make clean 1>>${test_output_file} 2>>${test_output_file}
make 1>>${test_output_file} 2>>${test_output_file}

for i in sample1 sample2 sample3 sample4
do
        make $i 1>>${test_output_file} 2>>${test_output_file}
        diff SnowProgramImp.java ../test/sample-
compiled/$i/SnowProgramImp.java >> /tmp/$i.diff
        if [ $? == 0 ]
        then
                echo "Test Status: TRANSLATION SUCCESS - Program : $i.snow"
        else
                echo "Test Status: TRANSLATION FAILURE - Program : $i.snow"
                echo "Diff File : /tmp/$i.diff"
        fi
done

unit_test_dir=../test/unit

for i in `ls ${unit_test_dir}`
do
        java com/google/code/pltsnow/gen/Parser ${unit_test_dir}/$i/tc.snow >
SnowProgramImp.java
```

```
        diff SnowProgramImp.java ${unit_test_dir}/$i/SnowProgramImp.java >
/tmp/$i.diff
        if [ $? == 0 ]
        then
                javac SnowProgramImp.java
                java SnowProgramImp > /tmp/$i.output
                diff /tmp/$i.output  ${unit_test_dir}/$i/tc.output >
/tmp/$i.diff
                if [ $? == 0 ]
                then
                        echo "Test Status: SUCCESS - Program : $i.snow"
                else
                        echo "Test Status: TRANSLATION SUCCESS; EXECUTION
FAILURE - Program : $i.snow"
                        echo "Diff File : /tmp/$i.diff"
                fi
        else
                echo "Test Status: TRANSLATION FAILURE - Program : $i.snow"
                echo "Diff File : /tmp/$i.diff"
        fi
done
cd ../test
```

8.2 Test Cases

The test suite for sn*w consists of the core sample programs specified in the language tutorial and unit test cases which are aimed at testing various sn*w constructs at the unit level. Some snippets of the test cases and the corresponding generated Java Code is shown in the table below.

8.2.1 If – Elseif - Else Construct

sn*w Code	Generate Java Code

`if i = 1 then`
`        print: "If Construct Works\n"`
`end`
	`if ((i.equals(new SnowAtom(1))).getInt() !=0 ){`
	
	`snw_print(new SnowAtom("If Construct Works\n"));`
	
	`}`

`if i = 0 then`
`        print: "If Construct does not work\n"`
	`if ((i.equals(new SnowAtom(0))).getInt() !=0 ){`
	
	`snw_print(new SnowAtom("If Construct does`

end	not work\n"));
	
	}

if i = 1 then
        print: "If-Else Construct Works\n"
else
        print: "If-Else Construct does not work\n"
end
	if ((i.equals(new SnowAtom(1))).getInt() !=0){
	
	snw_print(new SnowAtom("If-Else Construct Works\n"));
	
	}
	else {
	
	snw_print(new SnowAtom("If-Else Construct does not work\n"));
	
	}

if i = 1 then
        print: "If-Elseif Con-struct Works\n"
elseif j = 2 then
        print: "If-Elseif Con-struct does not work\n"
end
	if ((i.equals(new SnowAtom(1))).getInt() != 0){
	
	snw_print(new SnowAtom("If-Elseif Construct Works\n"));
	
	}
	else if ((j.equals(new SnowA-tom(2))).getInt() !=0){
	
	snw_print(new SnowAtom("If-Elseif Construct does not work\n"));
	
	}

8.2.2 For Construct

sn*w Code	Generate Java Code

for i from 1 to 10
        print: "For Construct Works " + i + " \n"
end
	for(i = new SnowAtom(1);!i.hasApproached(new SnowAtom(10),new Sno-wAtom(1));i.moveTowardsBy(new SnowA-tom(10),new SnowAtom(1),new SnowA-tom(1))){snw_print(new SnowAtom("For Con-struct Works ").plus(i).plus(new SnowAtom(" \n")));
	
	}

for i from 1 to 10 by 2
        print: "For Construct Works " + i + " \n"
end
	for(i = new SnowAtom(1);!i.hasApproached(new SnowAtom(10),new Sno-wAtom(1));i.moveTowardsBy(new SnowA-tom(10),new SnowAtom(2),new SnowA-tom(1))){snw_print(new SnowAtom("For Con-struct Works ").plus(i).plus(new SnowAtom(" \n")));
	
	}

for i from 10 to 1 by 1
        print: "For Construct Works " + i + " \n"
end
	for(i = new SnowAtom(10); !i.hasApproached(new SnowAtom(1),new SnowAtom(10));i.moveTowardsBy(new SnowAtom(1),new SnowAtom(1),new SnowAtom(10))){snw_print(new SnowAtom("For Construct Works ").plus(i).plus(new SnowAtom("\n")));
	
	}

for i from 10 to 1 by 2
        print: "For Construct Works " + i + " \n"
end
	for(i = new SnowAtom(10); !i.hasApproached(new SnowAtom(1),new SnowAtom(10));i.moveTowardsBy(new SnowAtom(1),new SnowAtom(2),new SnowAtom(10))){snw_print(new SnowAtom("For Construct Works ").plus(i).plus(new SnowAtom("\n")));
	
	}

8.2.3 While Construct

sn*w Code	Generate Java Code

**while i != 10
        print: "While Construct works " + i + "\n"
                i++
end**
	while((i.nequals(new SnowAtom(10))).getInt() != 0){
	snw_print(new SnowAtom("While Construct works ").plus(i).plus(new SnowAtom("\n")));
	i.increment();
	
	}

8.2.4 Operators

sn*w Code	Generate Java Code

**var val1 = i + j
if val1 = 3 then
        print: "Arithmetic + works\n"
else
        print: "Arithmetic + does not work\n"
end**
	SnowType val1 = i.plus(j);;
	if ((val1.equals(new SnowAtom(3))).getInt() !=0){
	
	snw_print(new SnowAtom("Arithmetic + works\n"));
	
	}
	else {
	
	snw_print(new SnowAtom("Arithmetic + does not work\n"));
	
	}

**var val2 = i - j
if val2 = 1 then
        print: "Arithmetic - works\n"**
	SnowType val2 = i.minus(j);;
	if ((val2.equals(new SnowAtom(1))).getInt() !=0){

else
print: "Arithmetic - does not work\n"
end
	snw_print(new SnowAtom("Arithmetic - works\n"));
	
	}
	else {
	
	snw_print(new SnowAtom("Arithmetic - does not work\n"));
	
	}

var val3 = k * j
if val3 = 2 then
print: "Arithmetic * works\n"
else
print: "Arithmetic * does not work\n"
end
	SnowType val3 = k.times(j);;
	if ((val3.equals(new SnowAtom(2))).getInt() !=0){
	
	snw_print(new SnowAtom("Arithmetic * works\n"));
	
	}
	else {
	
	snw_print(new SnowAtom("Arithmetic * does not work\n"));
	
	}

var val4 = k / i
if val4 = 1 then
print: "Arithmetic / works\n"
else
print: "Arithmetic / does not work\n"
end
	SnowType val4 = k.divide(i);;
	if ((val4.equals(new SnowAtom(1))).getInt() !=0){
	
	snw_print(new SnowAtom("Arithmetic / works\n"));
	
	}
	else {
	
	snw_print(new SnowAtom("Arithmetic / does not work\n"));
	
	}

var val7 = b ^ a * c
if val7 = 6 then
print: "Arithmetic Precedence 1 works\n"
else
print: "Arithmetic Precedence 1 does not work\n"
end
	SnowType val7 = b.power(a).times(c);;
	if ((val7.equals(new SnowAtom(6))).getInt() !=0){
	
	snw_print(new SnowAtom("Arithmetic Precedence 1 works\n"));
	
	}
	else {
	
	snw_print(new SnowAtom("Arithmetic Precedence 1 does not work\n"));
	
	}

val1 = "i" + "j"
if val1 = "ij" then
print: "String + works\n"
	val1.set(new SnowAtom("i").plus(new SnowAtom("j")));
	if ((val1.equals(new SnowA-

else
        print: "String + does not work\n"
end
	tom("ij"))).getInt() !=0){
	
	snw_print(new SnowAtom("String + works\n"));
	
	}
	else {
	
	snw_print(new SnowAtom("String + does not work\n"));
	
	}

var i = 0
var j = 1
 	
if i < j then
        print: "Relation Lesser Than works\n"
end
 	
if j < i then
        print: "Relation Lesser Than does not work\n"
end
	SnowType i = new SnowAtom(0);;
	SnowType j = new SnowAtom(1);;
	
	if ((i.lt(j)).getInt() !=0){
	
	snw_print(new SnowAtom("Relation Lesser Than works\n"));
	
	}
	
	if ((j.lt(i)).getInt() !=0){
	
	snw_print(new SnowAtom("Relation Lesser Than does not work\n"));
	
	}

i = "a"
j = "b"
if i < j then
        print: "Relation Lesser Than works\n"
end
 	
if j < i then
        print: "Relation Lesser Than does not work\n"
end
	i.set(new SnowAtom("a"));
	j.set(new SnowAtom("b"));
	if ((i.lt(j)).getInt() !=0){
	
	snw_print(new SnowAtom("Relation Lesser Than works\n"));
	
	}
	
	if ((j.lt(i)).getInt() !=0){
	
	snw_print(new SnowAtom("Relation Lesser Than does not work\n"));
	
	}

var i = 0
 	
i++
 	
if i = 1 then
        print: "Postfix ++ works\n"
end
 	
i--
 	
if i = 0 then
	SnowType i = new SnowAtom(0);;
	
	i.increment();
	
	if ((i.equals(new SnowAtom(1))).getInt() !=0){
	
	snw_print(new SnowAtom("Postfix ++ works\n"));
	
	}

`print: "Postfix --`
`works\n"`
`end`
	`i.decrement();`
	
	`if ((i.equals(new SnowAtom(0))).getInt() !=0 ){`
	
	`snw_print(new SnowAtom("Postfix -- works\n"));`
	
	`}`

`if i = 1 and  j = 0 then`
`print: "Logical And`
`works\n"`
`end`
	`if ((i.equals(new SnowA-`
	`tom(1)).log_and(j.equals(new SnowA-`
	`tom(0)))).getInt() !=0 ){`
	
	`snw_print(new SnowAtom("Logical And works\n"));`
	
	`}`

`if i = 1 or j = 0 then`
`print: "Logical Or`
`works\n"`
`end`
	`if ((i.equals(new SnowA-`
	`tom(1)).log_or(j.equals(new SnowA-`
	`tom(0)))).getInt() !=0 ){`
	
	`snw_print(new SnowAtom("Logical Or works\n"));`
	
	`}`

`if not (i = 0) then`
`print: "Logical Not`
`works\n"`
`end`
	`if (((i.equals(new SnowA-`
	`tom(0))).log_not()).getInt() !=0 ){`
	
	`snw_print(new SnowAtom("Logical Not works\n"));`
	
	`}`

9 Conclusions

9.1 Lessons Learned

As a whole, we learned the importance of teamwork. This project was far larger than any that we had attempted in other courses, and really required a lot of coordination. To those ends, we really benefited from using Google Code & Subversion. We also learnt the importance of acknowledging what you do and don't know: we originally planned to target C, then switched to Java midstream when we realized that our C knowledge was not sufficient to complete the project to our standards.

9.1.1 Jonathan

This project reinforced software engineering techniques that I learned last semester in Prof Kaiser's 4156 Advanced Software Engineering. Developing a project management plan early and sticking to it was almost as important as developing a grammar early ad sticking to it. By creating a work breakdown early on, and a process for getting to the final project, we were able to stay focused on the goal, yet remain incredibly detail oriented.

I also recognized the importance of maintaining open channels of communication within a group: if each of us knows exactly where the others stand, the team as a whole is stronger. When one team member gets bogged down and drops the rope, the others are able to pick it back up.

The project was a lot of fun, and I really enjoyed getting to see the fruits of my labors.

9.1.2 Willi

It was exceedingly important that the SNOW group defined its vision early on, and stuck to it throughout the project. This was especially visible through the grammar, which changed by only a single line over the many months. Having a stable and well defined foundation for discussions made debates not only more interesting, but more inclusive. As each member was required to understand the vision, the grammar, and the current state of implementation, weekly meetings moved along very quickly and produced generally high quality ideas.

It was interesting, when everyone had the same basis, how that basis was reinforced through the face to face debates. It was difficult for a single group member to suggest changes to the language specification, for there were three or four other members who only understood the initial document. For this reason, changes to the specification were required to be extremely persuasive and well reasoned. It is unlikely that a group with a differing sense purpose would be able to coherently organize updates to its core strategy.

Although documentation of individual code is taught as a requirement for good programming, it was enlightening to see its effect. There were sections of code that I may have written one night which looked entirely foreign another. Having explanations of each section greatly improved the efficiency at which one could develop, and freed up time for the inevitable bug fixing.

9.1.3 Vinay

1) Starting early is the best thing we did
2) We took a 2 way approach: Distributed minor tasks and did it individually and also pooled in ideas and did it together for complex tasks. This kind of team approach helped a lot.
3) If you have a great team, you can aim really high and achieve it too.
4) Learnt time management and project management strategies.

9.1.4 Cesar

As the system integrator I had to learn how to deal with everyone using different operating systems. So one of the things I did to simply this task was make a more robust makefile to deal with various operating systems and yet make it so no one had to learn any special make command for their operating system. Setting up a GUI environment for development with SVN is often a hard process as the compiled code and project specific files are often computer/user associated so when they are committed or updated conflicts tend to arise. I found that making SVN ignore these files is the easiest to reduce conflicts due to these files, instead of making sure no one in the group commits them. Overall the most important lesson I learnt is that one should be familiar with all the tools being used as you can not predict how everyone will use their systems, but you will need to be able to fix it.

9.1.5 Sharadh

I found this a very interesting project to work on and it have a lot of insight into the inside workings of a programming language and a compiler.

It is worth spending a lot of time ensuring that the grammar is correct. The grammar forms the core of the compiler and a correct grammar goes a long way in ensuring that development is smooth.

SVN and Google Code are great for version management. It is especially helpful when multiple members of team are in involved in the development.

Working in a team of talented individuals is an experience in itself.

Advice for future teams:
1. Start early.
2. Get your grammar ready asap.
2. Incremental development and testing is the best way to go about things.

9.2 Advice for future teams

To other teams, we'd suggest that you head Prof Aho's advice on names: first come up with a logo, then a name, THEN the language. We came up with the name "snow" literally the same day that we formed the team – long before we knew what our language would be. It helped us focus our domains, and sparked a few ideas that probably wouldn't have come out otherwise.

We would also suggest that future teams do some serious soul searching when deciding on what language to target to: we messed around for too long with C before switching to Java, and that was time lost. Along the same vein, it is vital to decide on a bulletproof grammar BEFORE IMPLEMENTING ANYTHING, and then STICK to that grammar. We did, and it sure made life easier.