NCML

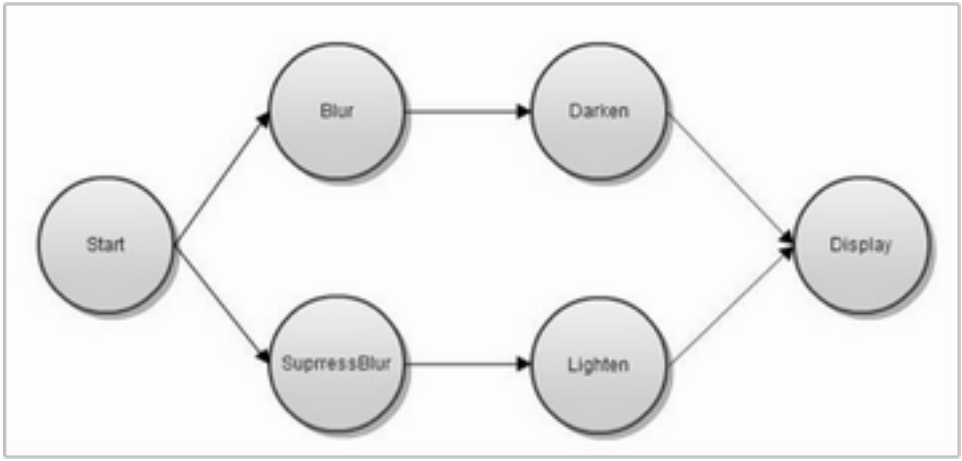
by Victor, Seshu, and Nina

1. Introduction

Motivation:

The motivation here was to create an expressive and natural language for expressing systems of “nodal computation,” which, broadly speaking, is a type of function composition from a graph theoretic perspective.

The idea came about as a result of Victor’s work in animation, where certain types of specialized software employ nodal computation as their paradigm of choice.



Project Proposal Document:

Team: Nina Berg (nb2555)
Victor Frenkel (vgf2103)
Venkata Yamajala (vsy2104)

About Nodal Computation:

The main idea behind this language is to allow **nodal computation (NC)**. NC is a way to design programs around nodes that contain data, a compute function, and input/output connections that define dependencies between nodes and flow of data.

A program in NodalCompML will contain a set of *StartNode*s, along with a series of regular *Nodes*. Each Node (including the StartNode) will have their own *compute* function. The output of each node then becomes the input to the next node. Program execution ends when the exit() function is called.

START NODES

We require an explicit set of start nodes that do not have dependencies on any other nodes. The start node may receive arguments from the command line. It then forwards its data to its adjacent nodes to start computation.

DATA PART OF NODE/ NODE STRUCTURE

Every node can contain an arbitrary number of local variables to be used by the compute function. A separate area is used to store the result of the compute function, to be distributed

to other nodes via output connections from the node. Nodes consist of a required *compute* function, as well as any necessary local variables or helper functions.

NODE I/O

The input and outputs of every node must be defined using the basic types given in NCML. (double, float, int, string, char, boolean, **but not the node type**). All compute functions accept a list of identified input parameters. The output of each compute function is made via the *forward* command. Data outputs are forwarded to the specified input parameter of a chosen node. The compiler will verify that these types match.

The various input parameters to a Node's compute function can come from multiple nodes. Additionally, nodes can selectively forward their data based on boolean conditions using if-else constructs. This functionality may result in a situation where a node does not receive all the inputs it needs, or received multiple inputs for the same input parameter.

Each forward command can send one output value to an input value. Multiple copies of the same data can be sent to different nodes. Forwards may also send the same value to multiple input parameters of the same node. Nodes can choose to forward data back to themselves recursively.

COMPUTE FUNCTION

The compute function accepts a list of input data, acts on the data, and forwards output to the next node in the program. The language allows for helper functions to be defined, breaking the compute function for each node into various pieces. However, all data i/o must be done within the compute function.

The function syntax will resemble C++/Java syntax, however, we will not be using semi-colons to end statements. Individual statements will be separated by a newline.

DEPENDENCIES/RE-EVALUATION

When a node receives a forwarded message from another node, its compute function is rerun on the changed field along with previous values for unchanged fields. If a node does not receive input for a field it simply blocks. If a node receives multiple inputs for a field, it computes using the most recent value for that field. This can occur in the case of conditional forwarding, where two nodes are inputting into the same field.

LANGUAGE FEATURES

The language will support creation of arrays out of any of the basic types (useful for working with images, matrices, etc.)

Problem Being Addressed

Nodal Computation, or rather DAGs, allow for thinking about computation from a different perspective. While O'Caml does not do anything that C cannot, it offers programmers a new,

and in some respects more elegant, perspective on pre-existing concepts. In the way O'Caml encourages a programmer to think about the function in a mathematically rigorous fashion, NodalCompML encourages the programmer to conceptualize the relationships between function applications in a visual way.

However, NodalCompML is not just an elegant way of thinking about function composition like algorithmic procedures, it is also a computational paradigm that has been heavily employed in the graphics processing applications.

Ultimately we believe that this language delivers a valuable and expressive way to describe certain types of serieses of operations, while at the same time being mathematically elegant.

Node Template::

```
Node <node_name> {

        // local variables

        fun compute(<type> <field_name> , ...) {
                float a
                int b

                //do work - If/else, while, do-while

                forward (a) to (<node>.<field_name>)
                forward (b) to (<node>.<field_name>)
        }

        <type> fun <function_name> (params...) {
                // do work

                return <type>
        }
}
```

Representative Program

```
Node StartNode {
        fun compute(int argc, string argv) {
                //We can do some error checking here
                forward (argv[1]) to (Read.filename)
                forward (argv[2]) to (Write.filename)
        }
}
```

```
Node Read {
	fun compute(string filename) {
		file f = open(filename)
		if (f != NULL)
			forward (f) to (Filter.f)
		else
			forward (“error opening file”) to (Error.message)
	}
}

Node Filter {
	fun applyFilter(file f) {
		// do some stuff to file
		return modifiedFile;
	}

	fun compute(file f) {
		file mf = applyFilter(f)
		forward (mf) to (Write.f)
	}
}

Node Write {
	fun compute(string filename, file f) {
		if write(filename, f)
			exit()
		else
			forward (“error writing file”) to (Error.message)
	}
}

Node Error {
	fun compute(string message) {
		print message
		exit()
	}
}
```

2. Language Tutorial

Compiling an ncml program: unzip the file into a local directory, type make. Pass in the “-ast” command line flag to compile using Python AST backend. Finally nun ./ncml <name of ncml program> to execute your ncml program. Requires an Ocaml compiler and Python Interpreter.

1. Nodes
    - All ncml programs must start with a node named start.

- node <name> (<optional parameter list>) {
  <node body>
  }
- node structure
  - node name is a unique identifier
  - a list of input parameters to the node
  - a list of variables local to the compute function
  - any helper functions
  - the implicit compute function

2. Comments
   - // until the end of the line
3. Types supported
   - int, float, string, bool, char, void
   - no type promotions or casting
4. The compute function
   - if/else, for, while - conditions must be boolean
   - break, continue -> i think these are supported, ask victor/seshu if they're implemented
   - forward statements - only allowed within a compute function
   - print function
   - All node parameters and compute local variables are visible here
5. Helper functions
   - fun <return type> <name> (<optional list of parameters>) { <body> }
   - name of the function must be unique within a node
   - All variables must be declared first
   - The only variables visible here are the function inputs and function local variables
   - Can call other helper functions
   - forwarding not allowed
   - return statements

6. Some sample programs:

Sample program: Decrementor

```
node start() {   //This is our start node
  int i = 5;    // i a local variable, only visible within compute

  while (i > 0) {
      print i;  // print function, prints the value of i to std out
      i = i - 1;
  }
  print 42;
  forward (i) to decrementor;  //forward statmene
}
node decrementor(int x) {  //Another node, it accepts x as an input
  int y = 2;
  print (x - y);
}
```

//I dont think we need this for the tutorial - Nina

Python code generated:

```
def forward(node, args):
  nodes[node].compute(*args)
def main():
      global nodes
  nodes = { 'decrementor':decrementor(),     'start':start() }
       nodes['start'].compute()
class decrementor(): # node name
       y = 2
       def compute(self, x):
              y = 2
              print x - y
class start():
       i = 5
       def compute(self, ):
           i = 5

           while(i > 0):
               i = i - 1
               print i
               print 42
               forward('decrementor', [i])
      main()
```

Sample program: Fibonacci

```
node start() {
fun int fib(int x) {
  if (x < 2) {
    return 1;
  }

  return fib(x-1) + fib(x-2);
}

print fib(0);
print fib(1);
print fib(2);
print fib(3);
print fib(4);
print fib(5);
}
```

Python code generated:

```
def forward(node, args):
  nodes[node].compute(*args)
```

```
def main():
  global nodes
  nodes = { 'start':start() }
  nodes['start'].compute()

class start(): # node name
  def fib(self, x):
      if (x < 2):
          return 1
          return self.fib(x - 1) + self.fib(x - 2)
      def compute(self, ):
          print self.fib(0)
          print self.fib(1)
          print self.fib(2)
          print self.fib(3)
          print self.fib(4)
          print self.fib(5)
main()
```

3. Language Manual

Language Reference Manual

Table of Contents

1 Introduction

NodalCompML is a language based on nodes processing data through a compute function. Nodes are connected to each other through inputs and outputs over which they forward data after having performed some computation on it.

1.1 The Node

NCML allows program computation to be divided among multiple nodes. Each node has a unique name, as well as a list of arguments it can accept. See section XX for details. An explicit start node, named start, is required.

The body of the node contains all computations related to that node including any local variable declarations, any local helper functions, as well as any forward statements. Forward statements are used to pass data between nodes. See section XX for more details on creating local helper functions. See section XX for more details on forwarding.

A node will begin its computation when it has received values for all of its inputs. If multiple values are received for one input, the most recently received value will be used. Note that input values may arrive from multiple nodes.

2 Lexical Conventions

2.1 Comments

The start of a comment is signified by // with the end being the end of the line on which the // appeared.

2.2 Identifiers

Identifiers are case sensitive sequences of letters and digits, in which the first character of the sequence must always be a letter.

2.3 Keywords

The following identifiers are reserved:

node	int	float
bool	string	if
else	while	for
fun	true	false
break	continue	return
forward	to	

2.4 Constants

Integer constants are sequences of digits. Double constants consist of sequences of digits a single . optional, and another sequence digits. String constants are sequences of letters and digits surrounded by double quotations “”. In order to use “ in a string it must be escaped by \. Boolean constants are true and false.

2.5 Operators

The following are reserved for operators:

+	-	*
/	==	!=
<=	>=	>
<	%	||
&&	!	

2.6 Punctuation

Statements and expressions do not have to be terminated by a semicolon. All that is required is if multiple expressions appear on a single line they be separated by commas.

3 Types

3.1 Primitive Types

NodalCompML supports the following basic types:

- char: Can hold a single 16 bit character. The character can be a letter, digit, punctuation, or control character.
- int: Signed 32-bit integer type. Can store any value in the range -2,147,483,648 to 2,147,483,647.
- float: single precision signed 32-bit floating point type.
- boolean: 1-bit data type with only two possible values, true or false.

3.2 Complex Types

- Array: Describes a collection of similar objects. Each element can be accessed by its location within the array. The size of the array must be declared when the array is first created. Upper limit on the size of the array is bound only by the amount of computing resources available. -> Not supported in current version
- String: Can hold a string of any size. Upper limit on the length of the string is bound only by the amount of computing resources available.

3.3 Type Conversion -> Not supported

Unary expressions of one type can be explicitly converted into another type by casting the expression as the desired type. Casting can be accomplished by preceding the expression with parenthesized name of the desired type.

Example: (type-name) cast-expression

Note that casting a more precise type into a less precise type may result in some data loss.

4 Expressions and Operators

4.1 Precedence and Associativity

The following table lists the precedence and associativity of NCML's operators. Operators in the same cell are of the same precedence and are associated as indicated. Rows are in order of decreasing precedence. Parentheses can be used to override precedence rules, as well as for clarity.

Operator	Description	Associativity
Primary Expression	Identifiers, Constants, Parenthesized Expressions	
()	Function calls	Left to right
+ - !	Unary plus and minus Logical NOT	Right to Left
* / %	Multiplication Division Modulus	Left to Right
+ - +	Addition Subtraction String Concatenation	Left to Right
< > <= >=	Relational Operators	Left to Right
== !=	Equality	Left to Right
&&	Logical AND	Left to Right
||	Logical OR	Left to Right

4.2 Primary Expressions

Primary expressions consist of identifiers, constants, string literals, or parenthesized expressions.

4.3 Function Calls

Function calls are made by calling the name of the function, followed by a parenthesized list of comma separated arguments. The argument list may be empty. The number of the arguments provided in the function call must match the number of arguments specified in the function definition. See Section XX for more details on function definitions.

4.4 Array References -> not supported

The [] operator can be used to access individual elements within an array. For example x[3] will return a reference to element 3 in array x. Note that array indices begin at 0. The expression within the square brackets must evaluate to an integer.
o

4.5 Unary Operators

+ or -: Results in the positive or negative value of the operand.
!: The result is the logical NOT of the operand.

4.6 Binary Arithmetic Operators

The following binary arithmetic operations are supported: multiplication (*), division (/), addition (+), subtraction (-), and modulus (%).
Note that the + operator is also used for string concatenation.

4.7 Relational and Equality Operators

The following relational operators are supported: less (<), greater (>), less or equal (<=), greater or equal (>=), equal (==), and not equal (!=). The result of these expressions is of types Boolean, indicating when the relation is true or false.

4.8 Logical Operators

The logical AND operator, &&, returns true if both its operands evaluate to true, otherwise false is returned.
The logical OR operator, ||, returns true if either of its operators evaluates to true, otherwise false is returned.
The logical NOT operator, !, is also supported. See section 4.5 for details.

4.9 Assignment

The = operator will replace the value of the left hand side with the evaluated expression on the right hand side.

5 Declarations

A declarator has exactly one decl-name specifying the identifier that is being declared.

SYNTAX:
declarator:
identifier

declarator [constant-expression[optional]]

5.1 Node Declaration

A node specifies a blackbox model for a computation to be performed. The node specifier defines a set of parameters to be received from other nodes, and parameters to be passed out along with which node they are being passed to. Input and output parameters can be given default values with the param = <default value> notation.

SYNTAX: -> needs to be updated

node-specifier:

node *declarator (argument-declaration-list[optional]) compound-statement*

argument-declaration-list:

argument-declaration-list, argument-declaration
argument-declaration

argument-declaration:

type-specifier declarator
type-specifier declarator = constant-expression

5.2 Interface Declaration -> not supported

An interface is a node-like way for libraries to expose complete pieces of functionality to users. The interface declares necessary inputs and output variables. Every interface declaration must have both inputs and outputs. The user then tells the library interface where to send its output by using a forward statement. Only after telling the interface where to forward its outputs can the user forward data into the inputs of the interface. In this way, we can create reusable third party libraries using NCML.

SYNTAX:

interface-specifier:

interface *declarator (argument-declaration-list) (argument-declaration-list)*

Here is an example of an interface being created in a library and used in a user program:

userprog.ncml

```
import mylib
import sys

Node my_comp_start1 ( ) {
        //take stuff here..
        image_filename = sys.argv[0]
        int[ ] in_image

        //various operations
```

```
        //define where output of interface goes
        forward (IComplicatedEffect.out_image) to (next_step.in_image)

        //kick off calculation by giving the interface its input.
        forward (in_image, 0.5, 3.7) to IComplicatedEffect.(in_image, param1, param2)
}

Node next_step (int[ ] in_image) {
        //do more stuff.
}
```

mylib.ncml

```
// declaring two interfaces here, IComplicatedEffect and IBlur
Interface IComplicatedEffect (int[ ] in_image, float param1, float param2) (int[ ] out_image) {
        forward (in_image, param1) to (CEStart1.image, CEStart1.param1)
        forward (in_image, param2) to CEStart2
}

// a simpler interface that exposes a run through one node in this library (blur).
Interface IBlur(int [ ] image, float strength) (int[ ] out_image) {
        forward (image, strength) to (Blur.image, Blur.strength)
}

Node Blur (int[ ] image, float strength) {
        int[ ] new_image = image
        //do stuff to image
        forward (new_image) to (IBlur.out_image)
}

Node CEStart1 (int [ ] image, float param1) {
        int [ ] new_img

        //do stuff to new_img

        forward (new_img) to op1
}

Node CEStart2 (int [ ] image, float param2) {
        int [ ] new_img

        //do stuff to new_img

        forward (new_img) to op2
```

```
}

// a bunch of nodes here in the sequence that eventually forward to CEFinalStep

Node CEFinalStep (int [ ] image, float param3) {
        forward IComplicatedEffect.out_img
}
```

5.3 Function Declaration

Although the statements in the body of a node encapsulate a computation, it is often desirable to break up the computation into smaller helper functions that can be called from the node's main code. Functions must be defined before the compute function begins

SYNTAX:

function-definition:

fun type-specifier function-declarator function-body

function-declarator:

declarator (argument-declaration-list)

function-body:

compound-statement

5.4 Variable Declaration

Variable declarations consist of a type specifier and the variable's declarator.

SYNTAX:

variable-declaration:

type-specifier declarator

6 Statements

6.1 If - Else Statements

NCML provide sthe standard if statement familiar from “C family” programming languages. Namely:

if (*expression*) *statement*

Where *statement* is executed only if *expression* evaluates to true. Similarly the usual if-else construction is provided:

if (*expression*) *statement1* else *statement2*

Where, as before, *statement1* is executed only if *expression* evaluates to true, and *statement2* is evaluated otherwise.

6.2 While Statements

Again NCML provides a standard while construction from the “C family,” with the usual format:

while (*expression*) *statement*
Where *statement* executed while *expression* evaluates to true. After every execution of *statement. expression* is evaluated to check if it is still true, in which case the loop continues, or false, in which case the loop terminates.

6.3 For Statements

Another classic “C family” construction is the for statement, which has the form

for (*expression-1opt* ; *expression-2opt* ; *expression-3opt*) *statement*

Which is equivalent to

```
expression-1
while ( expression-2 ) {
        statement
        expression-3
}
```

Where the first expression specifies initialization for the loop (this may not be necessary if the condition being tested in the second expression is defined within the correct scope, see below for scope rules), the second specifies the test that is conducted at the beginning of each iteration., and the third typically specifies the increment, performed after each iteration.

6.4 Break Statement

As in “C-family” languages the statement

break

causes the termination of the smallest enclosing while or for statement, hence ending the loop. Control is passed to the statement immediately following the terminated statement.

6.5 Continue Statement

Consistent with “C family” languages a continue statement is provided, where

continue

will pass control to the loop-continuation portion of the smalles enclosing while or for loop. More precisely in the following statement

```
while ( expression1 ) {
        statement1

        if ( expression2 ) {
                continue
        }

        statement2
}
```

If *expression2* evaulates to true, then *statement2* will be skipped, else if *expression2* evaluates to false *statement2* will be executed.

6.6 Return Statements

As in “C family” languages a return statement of the form

return *expression*

Specifies the return value of the function, however, such a statement is not required, and the absence of such a statement from the function is equivalent to return void. Note that return statements only make sense if the are called from within a helper function.

6.7 Forward Statements

Specific to NCML is the forward statement, which specifies where the result from a given Node is forwarded. More precisely, within the context (see scope) of myNode1, myNode1 may use a forwarding statement to forward any expression in that context to any Node, say myNode2. Thus the following expression forwards *expression* from myNode1, to myNode2

```
Node myNode1 () {
        forward ( expression ) to ( myNode2 )
}
```

Where myNode1 will forward the value of expression to an input parameter of myNode2, allowing code to be written in myNode2 that relies on the value, and type correctness of the value received in myNode2. This means there *cannot* be type errors for values that come from a Node's parameters.

7 Scope Rules

NCML is a lexically scoped language where identifiers cannot be declared outside Nodes. Hence the following is not syntactically correct, assuming the correct myNode2, definitions:

```
int x = 1;

Node myNode () {
        forward (x) to (myNode2)
}
```

Whereas the following is correct, again assuming presence of correct definitions for myNode2:

```
Node myNode () {
        int x = 1
        forward (x) to (myNode2)
}
```

While one may argue that this limits code reuse, and to a degree it certainly does, this scoping, or rather the combination of lexical scoping, and restricted location of identifiers forces the programmer to use the nodal computation paradigm, and structure their program in terms of Nodes, rather than just a few Nodes, and a whole litany of helper functions. This is consistent with the mission of the language: it offers a new way to think about certain types of computation.

4. Project Plan

Identify process used for planning, specification, development and testing

Concept creation: This being a school project, and not a profit seeking enterprise, the design discussion started at trying to build something that piqued the interests of all the team members. This started out with some kind of graph theoretic language. However, due to the fact that O'Caml already supports graph algorithms well, and the litany of quality graph libraries available, the team decided to pursue something tangentially related, but more original in motivation.

Planning: The team had a tight timeline in place towards the beginning to the semester, but as mentioned below, due to unforeseen changes in circumstances, this evolved. One particular act of planning that should be attributed to Nina was designating a "bare minimum" feature set to get the language implemented. Ultimately there was litany of features that we would have liked to have seen implemented, but by having a basic set of features that would ensure successful implementation was a great planning device, and ensured we got the project over the line in difficult circumstances.

Specification: Related to the above point about plan, our specification was more or less the minimum feature set that we implemented in the final codebase, which was well documented in the LRM. From the very beginning we were focused on being able to implement algorithms in our language.

Development: Our modular approach to development meant that development was able to proceed with little communication, however, this ultimately backfired. Seshu spent time developing a code generator to generate a Python AST, which was an excellent idea, and would have eased the implementation of some more advanced features, but ultimately was not feasible. Given the deadline we lost valuable time, and had to switch approaches. Regular updates to GitHub repositories, and use the testing suite, helped to ensure that broken code was not pushed to the main branch, hence team members were able to develop functionality independently.

Testing: Our testing strategy is outlined in detail in section six, but in a sentence it more or less amounted to implementing tests in a fashion similar to the “Micro C” compiler, with maximal code coverage, and requisite changes to reflect the idiosyncrasies of our language.

Include a one-page programming style guide used by the team

Strictly speaking the team did not actually use a programming style guide. However several factors made such guidelines partially redundant, and it was a conscious choice to allow team members stylistic freedom confident in the knowledge that the procedures that were put in place would make up for the lack of stylistic consistency.

The procedures followed in lieu of a strict style guide were:

1. Careful use of Git, and in particular an agreement from parties to push only code that compiles correctly, which combined with the use of O’Caml and the accompanying mathematical rigor is itself a strong quality control measure.
2. Before pushing code check that new failures were not introduced in the testing suite (see testing section for commentary on how this was implemented efficiently so that testing could easily be conducted on the code base with the addition of new code).
3. Modularity meant that for the most part team members were not working on the same files, and so in some sense it was not necessary to be intimately familiar with the code of other team members. The beauty of the interface between modules was that the interface negated the need for such familiarity.

Project timeline

Due to a disruption in the team, the majority of the development has taken place in the last two to three weeks. It has been much less organized than would have been preferable, and so beyond examining the logs, it is difficult to put a concrete timeline in place. Our original timeline went out the window rapidly.

Roles and responsibilities of each team member

NCML Top Level Executable

- Victor

Scanner
- Victor

Parser
- Victor

Code Generation (Python code version):
- Victor, Seshu

Code Generation (Python AST version):
- Seshu (note due to additional functions needed in the AST, this was a considerable undertaking).

Semantic Analyzer
- Nina

Abstract Syntax Tree:
- Nina, Seshu, Victor

Semantic AST:
- Nina

Tools and Development Environment:

Version control:
- We used Git, and the cloud hosting service GitHub. This allowed us to have the widely used, and well implemented, Git, and combined it with GitHub's cloud hosting service. This facilitated seamless

Programming languages:
- We used O'Caml for the front end of the compiler, and indeed code generation. The target language was Python.

Text Editors:
- Each team member used their own text editor, but the Emacs O'Caml syntax highlighting proved popular. It was possible to send O'Caml code directly to the O'Caml interpreter, which made for efficient inspection of an evolving code base.

Interpreter:
- The O'Caml interpreter proved useful for testing functions before actually putting them into code. Once It was known that a function passed the O'Caml compiler it was much less likely to break a large surrounding program.

Reports and Documentation:
- We used Google Docs to write all of the required documents. This allowed for collaborative editing, and seamless conversion to PDF.

Project log

For our project log we used the "Commit History" from Git. Since each commit is, at least in theory, accompanied by a descriptive message, and each commit should also correspond to an evolving of the project as a whole, this is a descriptive way to track the evolution of the project. The logs also provide plenty of detail.

```
commit ba37d09f7c0e35d5dd44d6f1b94d66b9254191c8
```

```
Author: Nina Berg <nb2555@columbia.edu>
Date:   Fri Dec 21 14:07:58 2012 -0500

   Modified the semantic tests to work slightly better with automated script
- note: these are supposed to fail

commit 31a946e49222d5e7c4755f799354d450e4f493c1
Author: Nina Berg <nb2555@columbia.edu>
Date:   Fri Dec 21 13:45:22 2012 -0500

   Added the semantics test programs, along with an automated script (based
off seshu's)

commit 1687854c442be69b0f7b2d8439b3672f1be4f327
Author: Nina Berg <nb2555@columbia.edu>
Date:   Fri Dec 21 12:48:43 2012 -0500

   Added some more checks to semantic

commit 179e1022711d0e4b55c29e9f22c757f8d87c5109
Author: Victor Greg Frenkel <vgf2103@columbia.edu>
Date:   Fri Dec 21 12:19:33 2012 -0500

   saving work.. tweaked code gen to fix gcd to no avail.

commit 26e96f01ff7e55f4822886e7769b5d9a20ad4e6c
Merge: 0e58974 0497b22
Author: Victor Greg Frenkel <vgf2103@columbia.edu>
Date:   Fri Dec 21 12:03:32 2012 -0500

   Merge branch 'master' of https://github.com/oscarbatori/NodalCompML

   Conflicts:
      src/semantic.ml

commit 0e589742237af5c015253c3e4f6cf1296a0e1af4
Author: Victor Greg Frenkel <vgf2103@columbia.edu>
Date:   Fri Dec 21 12:03:00 2012 -0500

   fixed if/if-else in code gen.

commit 0497b22f3455f4e70ff3bb8f73707befb3a0d66c
Author: Nina Berg <nb2555@columbia.edu>
```

```
Date:   Fri Dec 21 11:34:09 2012 -0500

   Removed some debug printouts in the semantic analyzer

commit 6054d008529fadf62b045307c359ab4aa80c3572
Author: Nina Berg <nb2555@columbia.edu>
Date:   Thu Dec 20 21:53:55 2012 -0500

   Turned on semantic analyzation in the git version of ncml.ml

commit 400642319eff779e6ec0088825dd36de49c48f55
Author: Nina Berg <nb2555@columbia.edu>
Date:   Thu Dec 20 21:33:36 2012 -0500

   Removed my garbage folder

commit b472f028e3cd6e633e16225762adb5aa879941dc
Author: Nina Berg <nb2555@columbia.edu>
Date:   Thu Dec 20 21:25:27 2012 -0500

   Cleaned up the makefile

commit 72d5fcb0d482a42e5cd7eb325c02670192b0d46c
Author: Nina Berg <nb2555@columbia.edu>
Date:   Thu Dec 20 21:25:04 2012 -0500

   Updated sast

commit 52ee64c7e2c149d0357182815c9fed563f608eb0
Author: Nina Berg <nb2555@columbia.edu>
Date:   Thu Dec 20 20:46:04 2012 -0500

   Fixed if/else bug

commit fc76fd0187a5cf19b910b810a1f3aa95b70d55d8
Author: Nina Berg <nb2555@columbia.edu>
Date:   Thu Dec 20 20:40:01 2012 -0500

   Cleaned up semantic analyzer, updated parts to support our simplified
scoping rules

commit 52343bbce8e93f0f6a13f30addfa141078e1581a
Author: Nina Berg <nb2555@columbia.edu>
```

```
Date:   Thu Dec 20 19:57:28 2012 -0500

   Semantic analyzer now recognizes forward statements

commit 988c2e8b7146c5710e72b71bc361b4e58f46253c
Author: Venkata Yamajala <vsy2104@columbia.edu>
Date:   Thu Dec 20 13:26:53 2012 -0500

   fixed small bug in ast backend

commit 077fab1adc29bfe150943d4ac913ee2454ec6265
Author: Venkata Yamajala <vsy2104@columbia.edu>
Date:   Thu Dec 20 12:16:28 2012 -0500

   fixed while loops

commit a9209a7549b0c8e6d0db9d04c8905c08c303dd40
Author: Venkata Yamajala <vsy2104@columbia.edu>
Date:   Thu Dec 20 09:32:17 2012 -0500

   script for testing ast backend

commit 33d27731819c0bb14575db37b16613a6a2efecd5
Author: Venkata Yamajala <vsy2104@columbia.edu>
Date:   Thu Dec 20 09:15:54 2012 -0500

   add old code gen as an optional cmd line switch

commit 58ebd00b00bc297a2fcd9803c4cab0a7e71a3925
Author: Venkata Yamajala <vsy2104@columbia.edu>
Date:   Thu Dec 20 02:57:55 2012 -0500

   no more fixed tabs. automated testing

commit 97e4f04fe8f1175dabab76309497bfb6c5ab9c16
Merge: 2f53427 e1e158d
Author: Nina Berg <nb2555@columbia.edu>
Date:   Tue Dec 18 20:35:44 2012 -0500

   Merge branch 'master' of https://github.com/oscarbatori/NodalCompML

commit 2f53427887f238352bdcee85935b02174c9098e1
Author: Nina Berg <nb2555@columbia.edu>
```

```
Date:   Tue Dec 18 15:29:46 2012 -0500

   Semantic Analyzer now supports function calls

commit e1e158d34693deb855c704745a5988e6693fd14e
Merge: be4d940 aec4b62
Author: Victor Greg Frenkel <vgf2103@columbia.edu>
Date:   Tue Dec 18 14:44:11 2012 -0500

   merged changes

commit be4d9401301ebc3cd25f60be9518430fdbf04e33
Author: Victor Greg Frenkel <vgf2103@columbia.edu>
Date:   Tue Dec 18 14:43:09 2012 -0500

   minor whitespace changes, saving before implementing tabify in code gen.

commit b2c8f80da1a75a2fc45e23445dfb26693c58913a
Author: Nina Berg <nb2555@columbia.edu>
Date:   Tue Dec 18 14:33:36 2012 -0500

   Semantic analyzer now supports helper functions

commit c7dd2cf727d60a32b6746c51b27cd94a86b75610
Author: Nina Berg <nb2555@columbia.edu>
Date:   Tue Dec 18 12:49:14 2012 -0500

   Semantic Analyzer updates

   Checks for start node, elseless if statments fixed, node parameters added
to node scope

commit aec4b62ac2de2749b9c62cebbf0983f03765f494
Author: Venkata Yamajala <vsy2104@columbia.edu>
Date:   Mon Dec 17 23:33:38 2012 -0500

   tests for semantic analyzer

commit 5c114392ac0e504e003c8cd74954188c829b3464
Author: Victor Greg Frenkel <vgf2103@columbia.edu>
Date:   Mon Dec 17 22:53:16 2012 -0500

   implemented data forwarding between nodes.
```

```
commit 0514f2b4d35820d2f429a214fbc1e182cb381d12
Author: Victor Greg Frenkel <vgf2103@columbia.edu>
Date:   Mon Dec 17 20:59:00 2012 -0500

   implemented code gen of while loops.

commit ff39d95f913bbb36d241541912de44e48d6721a2
Author: Victor Greg Frenkel <vgf2103@columbia.edu>
Date:   Mon Dec 17 19:35:10 2012 -0500

   fixed conversion of boolean to a string.

commit 8559bf82f8593964b67b9677edd1fdcc90ec1f52
Merge: c200d8e 0625a5a
Author: Victor Greg Frenkel <vgf2103@columbia.edu>
Date:   Mon Dec 17 19:10:59 2012 -0500

   Merge branch 'master' of https://github.com/oscarbatori/NodalCompML

commit c200d8e151dd98833ff9d8423bfe5adb722cfc1e
Author: Victor Greg Frenkel <vgf2103@columbia.edu>
Date:   Mon Dec 17 19:10:46 2012 -0500

   fixed up a bug in print with quotes, and added for loop implementation.

commit 0625a5a2672d1b3a275bd063bfadb25f8721a644
Author: Nina Berg <nb2555@columbia.edu>
Date:   Mon Dec 17 19:03:17 2012 -0500

   Fixed adding locals in semantic analyzer

commit 9f3d5af411b21b3487ff68dedef8f4a0435c2152
Author: Nina Berg <nb2555@columbia.edu>
Date:   Mon Dec 17 18:31:01 2012 -0500

   Added initial sast and semantic analyzer

commit 1b99296eec7f154b3ebaf7258373c774c6cb7bd4
Author: Victor Greg Frenkel <vgf2103@columbia.edu>
Date:   Mon Dec 17 00:55:03 2012 -0500

   reimplemented the entire code-gen phase of the compiler.
```

```
commit 31ecd786ad8b167e7d23ad66e96e4a6b9f3d46a2
Author: Victor Greg Frenkel <vgf2103@columbia.edu>
Date:   Sun Dec 16 17:17:29 2012 -0500

   fixed tests to use correct syntax.

commit ae743546efd505d49b436500b7a191ac65fd7e57
Merge: e46de3a 394e7f2
Author: Venkata Yamajala <vsy2104@columbia.edu>
Date:   Sun Dec 16 16:47:20 2012 -0500

   have makefile delete more junk when doing clean

commit e46de3a78d2177c9bbbc91b2eb26d6de3a1efe92
Author: Venkata Yamajala <vsy2104@columbia.edu>
Date:   Sun Dec 16 16:45:53 2012 -0500

   working variables in nodes

commit 394e7f2233a0b2c2105596e525f3f5fcddc50715
Author: Victor Greg Frenkel <vgf2103@columbia.edu>
Date:   Sun Dec 16 14:07:52 2012 -0500

   removing trash pyc files..

commit 396e06289780f6fc2a96b51e2d6ed5e2502a7f4b
Author: Victor Greg Frenkel <vgf2103@columbia.edu>
Date:   Sun Dec 16 14:07:33 2012 -0500

   integrated an ast pretty printing module into the main executable, for
easier debugging.

commit 478f16a46c2b2bf2e94f431e3d541315ae681114
Author: Venkata Yamajala <vsy2104@columbia.edu>
Date:   Sun Dec 16 11:51:44 2012 -0500

   fixed fib bug

commit a8747ba6c4e8114ee67ebbfcfae3d2bb68ca28c3
Author: Victor Greg Frenkel <vgf2103@columbia.edu>
Date:   Sun Dec 16 05:25:19 2012 -0500
```

```
   added BoolOp handling in Binops

commit 06021dd8dc3830ebbba478d708d09b39ae48c628
Author: Venkata Yamajala <vsy2104@columbia.edu>
Date:   Sun Dec 16 04:52:05 2012 -0500

   fixed some code gen bugs

commit 5d5c41631beb734b208af625fbab63505840090d
Merge: deab00d d019a7e
Author: Venkata Yamajala <vsy2104@columbia.edu>
Date:   Sun Dec 16 03:03:05 2012 -0500

   Merge branch 'master' of https://github.com/oscarbatori/NodalCompML

commit deab00d030b24b39fc3b10cda7470571c8087288
Author: Venkata Yamajala <vsy2104@columbia.edu>
Date:   Sun Dec 16 03:02:45 2012 -0500

   renamed nodes in tests

commit d019a7ee95c5bb5c2d15e7d847cf8c07da5aab43
Author: Victor Greg Frenkel <vgf2103@columbia.edu>
Date:   Sun Dec 16 02:56:44 2012 -0500

   added sending of ast to python file that has compilation instructions, and
automatically run the python interpreter on that file.

commit 9a8b0645ec9d596f1174cdb685bae48e223a15c8
Merge: 2484f9d c9b35a8
Author: Venkata Yamajala <vsy2104@columbia.edu>
Date:   Sun Dec 16 01:54:45 2012 -0500

   Merge branch 'master' of https://github.com/oscarbatori/NodalCompML

commit c9b35a8f36b58065c8416a01f4e3cdb5df980d2d
Author: Victor Greg Frenkel <vgf2103@columbia.edu>
Date:   Sun Dec 16 01:26:03 2012 -0500

   made ncml.ml stop eating the ast output and actually provide it for use by
others.

commit 2484f9d0a1c331e091a6417ae443f57c9bd3a59e
```

```
Merge: 548ec42 3b866f1
Author: Venkata Yamajala <vsy2104@columbia.edu>
Date:   Sat Dec 15 13:20:10 2012 -0500

   Merge branch 'master' of https://github.com/oscarbatori/NodalCompML

commit 548ec42dc45be8fdf74e7cbf043cef9412bf469b
Author: Venkata Yamajala <vsy2104@columbia.edu>
Date:   Sat Dec 15 13:20:01 2012 -0500

   started adding forwarding

commit 3b866f1bc7ce9a6c6a24b387d3963c00f938b6b5
Merge: d744293 f160a40
Author: Victor Greg Frenkel <vgf2103@columbia.edu>
Date:   Fri Dec 14 21:48:29 2012 -0500

   added commented out Boolop related code gen stuff, need better way to
handle various binops

commit d7442930fe8e9eaa64ae864559574e6d09c4fcf4
Author: Victor Greg Frenkel <vgf2103@columbia.edu>
Date:   Fri Dec 14 21:47:48 2012 -0500

   added commented out Boolop related code gen stuff, need better way to
handle various binops

commit f160a4020f80cc355c3ec1864ab4aed512a87663
Author: Venkata Yamajala <vsy2104@columbia.edu>
Date:   Fri Dec 14 20:13:29 2012 -0500

   Hello, World

commit 105c5b4df40c49b03858f122d21c35a9c1a92ddb
Author: Victor Greg Frenkel <vgf2103@columbia.edu>
Date:   Fri Dec 14 17:09:19 2012 -0500

   added the print statement to ncml grammar parser

commit 19451ee8a8057090382e9da8801033a4d2971ff9
Merge: 1e342b7 843eb27
Author: Venkata Yamajala <vsy2104@columbia.edu>
Date:   Fri Dec 14 16:22:10 2012 -0500
```

```
   example.py

commit 1e342b7308ad5937f32517b035a91bfecf04c091
Author: Venkata Yamajala <vsy2104@columbia.edu>
Date:   Fri Dec 14 14:50:45 2012 -0500

   Fixed some code gen stuff

commit 843eb278d60c07f6a50dcd9012a628854694f902
Author: Victor Greg Frenkel <vgf2103@columbia.edu>
Date:   Fri Dec 14 12:44:02 2012 -0500

   minor change to output debuggin message on var declaration.

commit 03b93acfe659985a2a083907a966a15a64b4a3f5
Author: Victor Greg Frenkel <vgf2103@columbia.edu>
Date:   Fri Dec 14 12:39:21 2012 -0500

   changed main ncml compiler executable to read in a file given as command
line argument

commit 0083ceb6831fb4b06163c3cdd3b1e12754ab0254
Author: Victor Greg Frenkel <vgf2103@columbia.edu>
Date:   Fri Dec 14 12:11:51 2012 -0500

   fixed up issue with shift reduce conflict in node sequence of func and var
decls. node parsing integration complete, the last major component of NCML.

commit 4c90a7009acc7a228fd57063d933021ace1de501
Author: Victor Greg Frenkel <vgf2103@columbia.edu>
Date:   Fri Dec 14 11:15:52 2012 -0500

   temporary fix for shift reduce conflicts within node body grammar..

commit 7ffbf040f9e0edde22c1a6ba33619252edcc600e
Author: Victor Greg Frenkel <vgf2103@columbia.edu>
Date:   Fri Dec 14 01:51:35 2012 -0500

   finished incorporating functions and testing them out... I don't think the
IF-ELSE type of statement is actually being hit properly...

commit cc2ed73cd6ca63a8b856145b2c54f24dace4f292
```

```
Merge: 4d896b1 a1353e7
Author: Victor Greg Frenkel <vgf2103@columbia.edu>
Date:   Wed Dec 12 23:52:08 2012 -0500

   added in function parsing, commented out stmt as there is an error there
that causes shift/reduce ambiguity

commit 4d896b1eb39e62b1d8c3992f5d2e785e253a65e3
Author: Victor Greg Frenkel <vgf2103@columbia.edu>
Date:   Wed Dec 12 23:17:40 2012 -0500

   reintegrated complete functions into the parser. checking for bugs

commit a1353e793a408ca513120b6e4b57e6d20b0f3466
Merge: ec0501c e336f9e
Author: Venkata Yamajala <vsy2104@columbia.edu>
Date:   Wed Dec 12 13:51:38 2012 -0500

   added string literals

commit ec0501c63c13a7e4bd69d643f706a5e7bd14da40
Author: Venkata Yamajala <vsy2104@columbia.edu>
Date:   Wed Dec 12 13:48:30 2012 -0500

   Reorganized code in ast so it builds.

commit 0c7a66c7e42daa4d794467273897f4183d56abcd
Merge: ce057da 802ac79
Author: Venkata Yamajala <vsy2104@columbia.edu>
Date:   Wed Dec 12 13:43:46 2012 -0500

   Merged new parser

commit e336f9ec201517d8cfb99911528e995883d40ec4
Author: Victor Greg Frenkel <vgf2103@columbia.edu>
Date:   Tue Dec 11 23:54:39 2012 -0500

   fixed string literals (completed adding support for them to parsing)

commit ce057dad8a7769b658b02ab5a978493bfd67bd13
Author: Venkata Yamajala <vsy2104@columbia.edu>
Date:   Tue Dec 11 22:22:15 2012 -0500
```

```
   generate classes for nodes

commit 3745de94d854cc4bb0705eb42610c5b6b06341f3
Author: Venkata Yamajala <vsy2104@columbia.edu>
Date:   Tue Dec 11 20:58:06 2012 -0500

   Semi working functions

commit 802ac79cc7e02dbe1ee042d2f32c1c5c49951e85
Author: Victor Greg Frenkel <vgf2103@columbia.edu>
Date:   Tue Dec 11 14:26:46 2012 -0500

   copying over the parser that has errors to parser_ref.mly and beginning
reconstruction from basic building blocks.

commit d97af9def306d364458254cfdf0d513d77f2fa60
Author: Victor Greg Frenkel <vgf2103@columbia.edu>
Date:   Tue Dec 11 14:26:27 2012 -0500

   copying over the parser that has errors to parser_ref.mly and beginning
reconstruction from basic building blocks.

commit 030742b98edbaa8629c9136b9b55837be19f397d
Author: Venkata Yamajala <vsy2104@columbia.edu>
Date:   Sun Dec 9 02:06:51 2012 -0500

   Started working on code generation

commit 0ffef5d82746a8903aef724312054042adac0e59
Author: Seshu Yamajala <syamajala@gmail.com>
Date:   Mon Dec 3 21:18:35 2012 -0500

   Added tests from microc

commit b845da701d7ac5eb3d27da69700423be7272771c
Author: Nina Berg <nb2555@columbia.edu>
Date:   Sat Dec 1 17:46:06 2012 -0500

   Updated ast, parser and scanner. Now handles nodes, and most if/else,
while... constructs

commit fac93408c612833b40b18ace305c65f3827d64be
Author: Victor Greg Frenkel <vgf2103@columbia.edu>
```

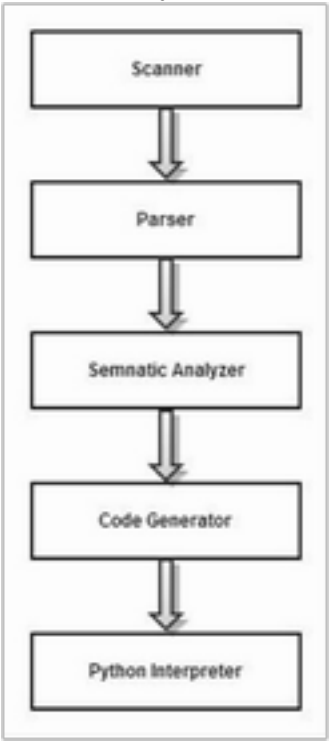
```
Date:   Sat Dec 1 17:29:07 2012 -0500

   prepping code generation

commit 337dfb06b1be3505073100e2aac40ef1bc83ed16
Author: Victor Greg Frenkel <vgf2103@columbia.edu>
Date:   Sat Dec 1 16:01:18 2012 -0500

   renamed parser_decls and added the rest of files to main src dir

commit 255be4c244f159e6723f4c803d2173f899098707
Author: Nina Berg <nb2555@columbia.edu>
Date:   Sat Dec 1 15:53:02 2012 -0500

   Moved parser.mly to main directory

commit cbc921ae30bb8290da017ad948860f718ec5ddb2
Author: Victor Greg Frenkel <vgf2103@columbia.edu>
Date:   Sat Dec 1 15:45:24 2012 -0500

   initial commit of generator source file

commit 25ffaac509468ff7e80660e344ce324a39e1c3ae
Author: Victor Greg Frenkel <vgf2103@columbia.edu>
Date:   Sat Dec 1 15:37:47 2012 -0500

   added the binops and unops to ast and parser_decls.

commit 45aea563b750e150d4cf5317c289f3fe176524ea
Author: Victor Greg Frenkel <vgf2103@columbia.edu>
Date:   Sat Dec 1 13:23:30 2012 -0500

   adding in binops.

commit 54c1a11a7e2b99a99b55c97a1facfd1751b21f96
Author: Victor Greg Frenkel <vgf2103@columbia.edu>
Date:   Sat Nov 24 14:03:09 2012 -0500

   integrating more parts... trying to fix 1 shift reduce conflict in the
parser_decls file. which should really just be renamed to parser because it's
now hacking on everything

commit 264b0e7fcf7cd7031a1fe2e75682b21e1df7ca5d
```

```
Author: Nina Berg <nb2555@columbia.edu>
Date:   Tue Nov 20 00:07:27 2012 -0500

   Added initial scanner.mll

commit 79806039875cc103bbd5fea470b4ddad64c7c0b2
Author: Victor Greg Frenkel <vgf2103@columbia.edu>
Date:   Mon Nov 12 22:09:29 2012 -0500

   finished a makefile using the desired ncml files.

commit 82731a715aaedeed308ce8bd210bcdb592f54f56
Author: Victor Greg Frenkel <vgf2103@columbia.edu>
Date:   Mon Nov 12 17:55:18 2012 -0500

   playing around with example reverse polish notation calculator parser

commit 16aa68aa000c62f4cbb1bcc4b30f42ce1bfb654d
Author: Victor Greg Frenkel <vgf2103@columbia.edu>
Date:   Mon Nov 12 16:24:50 2012 -0500

   initial commit

commit 179c12ac3ae811edc14a986bbdda2a84f7f0773d
Author: Victor Greg Frenkel <vgf2103@columbia.edu>
Date:   Mon Nov 12 16:24:25 2012 -0500

   initial commit
```

5. Architectural Design

Block Diagram:

The block diagram below illustrates the architecture of our compiler. All steps are in O'Caml, apart from the Python code that is ultimately generated by the Code Generator, which is subsequently interpreted



Interfaces:

As expected the scanner produces tokens, rejecting any file that contains invalid tokens. i.e. tokens that are not specifically accepted by our language. The parser then accepts the valid token stream that is output by the scanner, and puts them together while verfifying that the syntax is accepted by our language. The parser produces an AST, specified in ast.ml. The semantic analyzer is then passed the AST produced by the parser, and decides whether to accept/reject that AST. Amongst other things it verifies that types are correct, and that the referenced identifiers are legal. Provided the semantic analyzer accepts the AST for the given program, that AST is then passed to the code generator, which uses the structure of the AST, and the information contained in it, to generate Python bytecode. Finally the program is run by running the output Python program on the Python interpreter.

Assignment of Modules:

NCML Top Level Executable
- Victor

Scanner
- Victor

Parser
- Victor

Code Generation (Python code version):

- Victor

Code Generation (Python AST version):

- Seshu (note due to additional functions needed in the AST, this was a considerable undertaking).

Semantic Analyzer

- Nina

Abstract Syntax Tree:

- Nina, Sesu, Victor

Semantic AST:

- Nina

6. Test Plan

Representative Programs:

Program 3:
NCML:

```
node start() {
 int i = 5;

 fun int dec(int i) {
   return i-1;
 }

 while (i > 0) {
   print i;
   i = dec(i);
 }

 print 42;
 print i;
}
```

Python:

```
def forward(node, args):
        nodes[node].compute(*args)

def main():
        global nodes
        nodes = { 'start':start() }
        nodes['start'].compute()

class start(): # node name
       i = 5
       def dec(self, i):

               return i - 1

       def compute(self):
               i = 5

               while(i > 0):
                       print i
                       i = self.dec(i)
               print 42
               print i

main()
```

Program 2:

NCML:

```
node start() {
 int i = 0;
 string msg = "now_printing...";

 fun void foo() {
   int i = 0;
   for (i = 0 ; i < 10 ; i = i + 1) {
     print i;
   }
 }

 print 42;
 foo();
}
```

Python:

```
def forward(node, args):
        nodes[node].compute(*args)

def main():
        global nodes
        nodes = { 'start':start() }
        nodes['start'].compute()

class start(): # node name
       i = 0
       msg = "now_printing..."
       def foo(self):
               i = 0
               i = 0
               while(i < 10):
                       print i
                       i = i + 1

       def compute(self):
               i = 0
               msg = "now_printing..."
               print 42
               self.foo()

main()
```

Program 3:

NCML:

```
node start() {
 if (0 > 2)
```

```
    print 42;
  else
    print 8;
  print 17;
}
```

Python:

```
def forward(node, args):
        nodes[node].compute(*args)

def main():
        global nodes
        nodes = { 'start':start() }
        nodes['start'].compute()

class start(): # node name
       def compute(self):

               if (0 > 2):
                       print 42
               else:
                       print 8
               print 17

main()
```

Test Case Selection Procedure:

The main idea was to achieve the highest code coverage possible, and to this end the team decided to look at the Micro C compiler, and take cues from the testing suite offered up with it. To this end we attempted to test all of the basic operations of the language to the extent that the more sound the building blocks of a language are, i.e. they behave in the expected way, then composing those building blocks into algorithms is more likely to produce code that behaves in the expected wa.

Due to our abandoned Python AST generation approach we ended up having a tester for the AST approach, and for the approach that just generated regular Python code. Furthermore we tested the semantic analyzer, and code generation separately. That is we have a test suite that tests whether the code generator takes basic operations, and simple algorithms composed from them, and creates correct Python code that produces the expected output. These tests are NCML files. However, we also have a test suite for checking that the semantic analyzer rejects purposefully incorrect programs in the hope that we have enough cases to avoid the major pitfalls.

Automation Strategies:

Two automation strategies were employed. The first was to automatically compare output of test suite, to expected output, which was previously computed by hand. This created considerable time savings during testing. The second was to create a file that could run all of the tests at one time, so that any changes could instantly have their effect on the code base examined. This came in the form of a shell script tested “testastbackend.sh.”

Roles in Testing:

Seshu wrote most of the initial testing suite, and this was later supplemented by help from Victor. Seshu also wrote the file that automated the tests. Naturally all team members played some role in testing, as it touched on the work of every module in the compiler.

Test Suite:

test-arith1.ncml

```
node start() {
 print 39 + 3;
}
```

test-arith2.ncml

```
node start() {
 print 1 + 2 * 3 + 4;
}
```

test-fib.ncml

```
node start() {

 fun int fib(int x) {
   if (x < 2) {
   return 1;
     }
   return fib(x-1) + fib(x-2);
 }

 print fib(0);
 print fib(1);
 print fib(2);
 print fib(3);
 print fib(4);
 print fib(5);
}
```

test-for1.ncml

```
node start() {
 int i = 0;
 for (i = 0 ; i < 10 ; i = i + 1) {
   print i;
 }
 print 42;
}
```

test-for2.ncml

```
node start() {
 int i = 0;
 for (i = 0 ; i < 10 ; i = i + 1) {
   if (666 > 66) {
     print i;
   }
 }
 print 42;
}
```

test-for3.ncml

```
node start() {
 int i = 0;
 string msg = "now_printing...";

 fun void foo() {
   int i = 0;
   for (i = 0 ; i < 10 ; i = i + 1) {
     print i;
   }
 }

 print 42;
 foo();
}
```

test-forward1.ncml

```
node start() {
 int i = 5;
 while (i > 0) {
   print i;
   i = i - 1;
 }
 print 42;

 forward (i) to decrementor;
}

node decrementor(int x) {
 int y = 2;

 print (x - y);
}
```

test-func1.ncml

```
node start() {
 int a = 0;

 fun int add(int a, int b) {
   return a + b;
 }
```

```
 a = add(39, 3);
 print a;
}
```

test-func2.ncml

```
// Bug noticed by Pin-Chin Huang
node start() {
 int i = 1;

 fun int func(int x, int y) {
     print x+y;
 }

 func(2, i+1);
 print i;
}
```

test-func3.ncml

```
node start() {
 fun void printem(int a, int b, int c, int d) {
   print a;
   print b;
   print c;
   print d;
 }

 printem(42,17,192,8);
}
```

test-func4.ncml

```
node start() {
 fun int if_in_func() {
   if (3 > 1) {
     print 1;
   }
   return 666;
 }

 if_in_func();

}
```

test-gcd.ncml

```
//this one will fail, need to fix the if-else parsing.
node start() {
 fun int gcd(int a, int b) {
   while (a != b) {
     if (a > b) a = a - b;
     else b = b - a;
   }
   return a;
```

```
 }

 print gcd(2,14);
 print gcd(3,15);
 print gcd(99,121);
}
```

test-hello.ncml

```
node start() {
 print "hello_world";
 print 42;
 print 71;
 print 1;
}
```

test-if1.ncml

```
node start() {
 if (2 > 1) {
   print 42;
}
 print 17;
}
```

test-if2.ncml

```
node start() {
 if (2>1)
   print 42;
 else
   print 8;

 print 17;
}
```

test-if3.ncml

```
node start() {
 if (0 > 2)
   print 42;
 print 17;
}
```

test-if4.ncml

```
node start() {
 if (0 > 2)
   print 42;
 else
   print 8;
 print 17;
}
```

test-if5.ncml

```
node start() {
    int a = 42;
```

```
    float b = 3.14;

    if (4 > 2) {
      print b;
    }

    if ((5%4) == 1) {
      print "hello";
    }

    print a;
}
```

test-nestedif.ncml

```
node start() {
    int a = 42;
    float b = 3.14;

    if (4 > 2) {
         if ((5%4) == 1) {
           print "hello";
         }
//       print b;
    }


    print a;
}
```

test-ops1.ncml

```
node start() {
 print 1 + 2;
 print 1 - 2;
 print 1 * 2;
 print 100 / 2;
 print 99;
 print 1 == 2;
 print 1 == 1;
 print 99;
 print 1 != 2;
 print 1 != 1;
 print 99;
 print 1 < 2;
 print 2 < 1;
 print 99;
 print 1 <= 2;
 print 1 <= 1;
 print 2 <= 1;
 print 99;
 print 1 > 2;
 print 2 > 1;
 print 99;
 print 1 >= 2;
```

```
 print 1 >= 1;
 print 2 >= 1;
}
```

test-var1.ncml

```
node start() {
 int a = 4;
 int b = 3;
 int c = 2;
 int d = 1;
 a = 42;
 print a;
}
```

test-while1.ncml

```
node start() {
 int i = 5;

 fun int dec(int i) {
   return i-1;
 }

 while (i > 0) {
   print i;
   i = dec(i);
 }

 print 42;
 print i;
}
```

7. Lessons Learned

Nina:
I worked on the semantic analyzer. When I first started building the module, I tried to incorporate as many of the program features as possible, which resulted in a huge debugging headache. Once I broke it down and started adding one feature at a time, things went a lot smoother. I also discovered that using similar name in different modules, such as in the Ast and Sast can become extremely confusing, especially in a language with inferred typing.

Communication is key. Dividing up the work and then each member going off on their own to do the work is not the best way to get this project done. As changes are made to the design, especially the overall language design, discuss it as a group and keep all members updated. Redesigning various interfaces, while not desirable, is very feasible if it is discussed early on.

Seshu:
I learned a lot about the python ast and how compliation works in python. On the non technical side, when working on the original code generator, I learned that sometimes you need to give up on an idea, but that it can be really hard to know when to give up on it, especially after you've put a lot of work into it. Also, while working on the new code generator, I learned that things that you initially thought were hard to do can sometimes be really really easy, but if you don't sit down and try to solve a problem because you think its hard you may end up doing much more work than necessary.

I think communication among team members is important. I did not know what other members of our team were doing and this influenced certain technical decisions deeply, such as the decision to not make changes to the AST at all for fear of breaking it for someone else in the team. This led to jumping through a lot of hoops in the original code generator that could have easily been avoided by changing the AST.

Victor:
I worked on all parts of the compiler with the exception of the semantic analyzer, but I did look at how the semantic code worked. It was interesting to see how all the different pieces of the compiler connected together to transform and keep track of elements in the representation we came up with to the final generated python code/ast. I found that it's near impossible to have team members work on separate parts of the compiler without requiring constant communication and updates on what is changing. Although we tried to stick with the agreed upon interfaces for as long as possible, it was sometimes necessary to break other's people code and ask them to rewrite portions in order to keep progressing on the project.

Be sure to understand the concepts required for the project early. Be comfortable with quickly prototyping ideas and throwing them away before choosing a wrong direction and spending way too much time on it. Communicate constantly about any and all changes made to the specification of the language/how things are done so everyone knows what and how they need to develop different modules for the language. If someone's code must be broken for the sake of

making the design of the compiler better, do it. It will probably take less time for them to fix their code than to continue trying to come up with hacks to work around problems.

8. Appendix: Code Listings

ast.ml:

```
type binop = Add | Sub | Mult | Div | Mod | Eq | Neq | Lt | Leq | Gt | Geq | And | Or
type unop = Not | Neg

type dtype = CharType | StringType | IntType | FloatType | BoolType | VoidType

type formal = Formal of dtype * string

type expr =
    Id of string
  | CharLiteral of char
  | StringLiteral of string
  | IntLiteral of int
  | FloatLiteral of float
  | BoolLiteral of bool
  | Binop of expr * binop * expr
  | Unop of unop * expr
  | Assign of string * expr
  | Call of string * expr list
  | Noexpr

type var_decl = VarDecl of dtype * string * expr

type stmt =
    Block of stmt list
  | Expr of expr
  | Return of expr
  | If of expr * stmt * stmt
  | For of expr * expr * expr * stmt
  | While of expr * stmt
  | Forward of expr list * string
  | Print of expr
  | Continue
  | Break
  | Nostmt

type boolop = {
  left : expr;
  bop : binop;
  right : expr;
}

type compare = {
  left : expr;
  cmpop : binop;
  right : expr;
}

type fun_decl = {
  return_type : dtype;
  fname : string;
  formals : formal list;
  locals : var_decl list;
```

```
  body : stmt list;
}

type node = {
       nname : string;
       args : formal list;
       local_vars : var_decl list;
       compute : stmt list;
       functions : fun_decl list;
}

type program = node list

let attributesref = ref []

let rec string_of_expr = function
    CharLiteral(l) -> "Str('" ^ Char.escaped(l) ^ "')"
  | StringLiteral(l) -> "Str('" ^ l ^ "')"
  | IntLiteral(l) -> "Num(" ^ string_of_int(l) ^ ")"
  | FloatLiteral(l) -> "Num(" ^ string_of_float(l) ^ ")"
  | BoolLiteral(l) -> "Bool(" ^ string_of_bool(l) ^ ")"
  | Id(s) -> (if List.mem s !attributesref then
      "Attribute(Name('self', Load()), '" else
      "Name('") ^ s ^ "', Load())"
  | Binop(e1, o, e2) ->
      (match o with
      |Add|Sub|Mult|Div|Mod -> "BinOp(" ^ string_of_expr e1 ^ ", " ^
        (match o with
        | Add -> "Add()"
        | Sub -> "Sub()"
        | Mult -> "Mult()"
        | Div -> "Div()"
        | Mod -> "Mod()") ^ ", " ^ string_of_expr e2 ^ ")"
      |Eq|Neq|Lt|Leq|Gt|Geq -> "Compare(" ^ string_of_expr e1 ^ ", [" ^
        (match o with
        | Eq -> "Eq()"
        | Neq -> "NotEq()"
        | Lt -> "Lt()"
        | Leq -> "LtE()"
        | Gt -> "Gt()"
        | Geq -> "GtE()") ^ "], [" ^ string_of_expr e2 ^ "])"
      |And|Or -> "BoolOp(" ^
        (match o with
           | And -> "And()"
        | Or -> "Or()") ^ string_of_expr e1 ^ ", " ^ string_of_expr e2 ^ "]" )
  | Unop(o, e1) ->
    "UnaryOp(" ^
      (match o with
        Not -> "Not()"
        | Neg -> "USub()")  ^ string_of_expr e1 ^ ")"
  | Assign(v, e) -> (if List.mem v !attributesref then
      "Assign([Attribute(Name('self', Load()), '" ^ v ^ "', Store())], "  else
      "Assign([Name('" ^ v ^ "'), Store()], ") ^  string_of_expr e ^ ")"
```

```
  | Call(f, el) ->
    "Call(Attribute(Name('self', Load()), '" ^ f ^ "', Load()), [" ^ String.concat ", "
(List.map string_of_expr el) ^ "], [], None, None)"
  | Noexpr -> ""

let string_of_compare compare =
  "Compare(" ^ string_of_expr compare.left ^ ", [" ^
    (match compare.cmpop with
        Eq -> "Eq()" | Neq -> "NotEq()"
      | Lt -> "Lt()" | Leq -> "LtE()"
      | Gt -> "Gt()" | Geq -> "GtE()" )
      ^  "], [" ^ string_of_expr compare.right ^ "])"

let build_compare = function
  | Binop(e1, o, e2) -> { left=e1; cmpop=o; right = e2 }

let rec string_of_stmt = function
    Block(stmts) ->
       String.concat ", " (List.map string_of_stmt stmts)
  | Expr(expr) -> string_of_expr expr;
  | Return(expr) -> "Return(" ^ string_of_expr expr ^ ")"

| If(e, s, Block([])) -> "If(" ^ string_of_compare (build_compare e) ^ ", [" ^ string_of_stmt
s ^ "], [])"
  | If(e, s1, s2) ->  "If(" ^ string_of_compare (build_compare e) ^ ", [" ^
      string_of_stmt s1 ^ "], " ^ " [" ^ string_of_stmt s2 ^ "])"
  | For(e1, e2, e3, s) -> string_of_expr e1 ^ string_of_stmt (While(e2, Block([s; Expr(e3)])))
  | While(e, s) -> "While(" ^ string_of_compare (build_compare e) ^ "[" ^ string_of_stmt s ^
"], [])"
  | Print(expr) -> "Print(None, [" ^ string_of_expr expr ^ "], True)"
  | Break -> "Break"
  | Continue -> "Continue"
  | Nostmt -> ""

let string_of_dtype = function
    CharType -> "char"
  | StringType -> "string"
  | IntType -> "int"
  | FloatType -> "float"
  | BoolType -> "bool"
  | VoidType -> "void"

let string_of_vdecl = function
    VarDecl(x, y, z) -> string_of_expr (Assign(y,z))

let string_of_formal = function
    Formal(x, y) -> y

let string_of_fdecl fdecl =
  "FunctionDef('" ^ fdecl.fname ^ "', arguments([Name('self', Param())" ^
    (if fdecl.formals = [] then "]" else
```

```
(List.fold_left (fun x y -> x ^ ", " ^ y) "" (List.map (fun x -> "Name('" ^ string_of_formal x
^ "', Param())") fdecl.formals) ^ "])) ^ ", None, None, []), ["
     ^ (if fdecl.locals = [] then "" else (String.concat ", " (List.map string_of_vdecl
fdecl.locals))  ^ ", ")
     ^ (String.concat ", " (List.map string_of_stmt fdecl.body)) ^ "], [])"

let string_of_node_vars = function
  | VarDecl(t, v, e) -> v

let string_of_compute n =
  string_of_fdecl ({ return_type = VoidType; fname = "compute"; formals = n.args; locals =
n.local_vars; body = n.compute })

let string_of_node ndecl =
  attributesref := (List.map string_of_node_vars ndecl.local_vars);

"ClassDef('" ^ ndecl.nname ^ "', [], " ^ "[" ^ string_of_compute ndecl ^ ", " ^ String.concat
", " (List.map string_of_fdecl ndecl.functions) ^ "], [])"

let string_of_program nodes =

"Module([Assign([Name('nodes', Store())], Dict([], [])), FunctionDef('forward',
arguments([Name('node', Param()), Name('args', Param())], None, None, []),
[Expr(Call(Attribute(Subscript(Name('nodes', Load()), Index(Name('node', Load())), Load()
), 'compute', Load()), [], [], Name('args', Load()), None))], []), " ^ (String.concat ", "
(List.map string_of_node nodes))
  ^ ", FunctionDef('main', arguments([], None, None, []), [Expr(Call(Attribute(Call(Name('" ^
(List.hd nodes).nname
  ^ "', Load()), [], [], None, None), 'compute', Load()), [], [], None, None))], []),
Expr(Call(Name('main', Load()), [], [], None, None))" ^ "])"
```

generator.ml:

```
open Ast
open Printf

let tab lvl =
  String.make lvl '\t'

let str_of_bool = function
  | true -> "True"
  | false -> "False"

let rec str_of_expr = function
  | CharLiteral(l) -> Char.escaped(l)
  | StringLiteral(l) -> l
  | IntLiteral(l) -> string_of_int(l)
  | FloatLiteral(l) -> string_of_float(l)
  | BoolLiteral(l) -> str_of_bool(l)
  | Id(s) -> s
  | Binop(e1, o, e2) ->
    str_of_expr e1 ^ " " ^
      (match o with
```

```
          Add -> "+" | Sub -> "-"
        | Mult -> "*" | Div -> "/" | Mod -> "%"
        | Eq -> "==" | Neq -> "!="
        | Lt -> "<" | Leq -> "<=" | Gt -> ">" | Geq -> ">="
        | Or -> "or" | And -> "and" )
    ^ " " ^ str_of_expr e2
  | Unop(o, e) ->
    (match o with
       Not -> "not" | Neg -> "-") ^ str_of_expr e
  | Assign(v, e) -> v ^ " = " ^ (str_of_expr e)
  | Call(f, e) -> "self." ^ f ^ "(" ^ (String.concat ", " (List.map str_of_expr e)) ^ ")"
  | Noexpr -> ""

let rec str_of_stmt s lvl =
  match s with
  | Block(stmts) -> (let l = "\n"^(tab lvl) in

(String.concat l (List.map (fun x-> str_of_stmt x (lvl+1)) (List.rev(stmts)))))
  | Expr(expr) -> str_of_expr expr;
  | Return(expr) -> "return " ^ str_of_expr expr
  | If(e, s, Nostmt) -> "if (" ^ str_of_expr e ^ "):\n" ^ (tab (lvl+1)) ^ str_of_stmt s
(lvl+1)
  | If(e, s1, s2) -> "if (" ^ str_of_expr e ^ "):\n" ^ (tab (lvl+1)) ^ str_of_stmt s1 (lvl+1)
^ "\n" ^ (tab (lvl)) ^ "else:\n" ^ (tab (lvl+1)) ^ str_of_stmt s2 (lvl+1)
  | For(e1, e2, e3, s) -> str_of_expr e1 ^ "\n" ^ (tab lvl) ^ "while(" ^ str_of_expr e2 ^
"):\n" ^ (tab (lvl+1)) ^ str_of_stmt s (lvl+1) ^ "\n" ^ (tab (lvl+1)) ^ str_of_expr e3
  | While(e, s) -> "\n" ^ (tab lvl) ^ "while(" ^ str_of_expr e ^ "):\n" ^ (tab (lvl+1)) ^
str_of_stmt s (lvl+1)
  | Forward(elist, dest) -> "\n" ^ (tab lvl) ^ "forward('" ^ dest ^ "', [" ^ (String.concat
"," (List.map str_of_expr elist)) ^ "])"
  | Print(expr) -> "print " ^ (str_of_expr expr)
  | Break -> "break"
  | Continue -> "continue"
  | Nostmt -> ""

let str_of_vdecl v lvl =
  match v with
  | VarDecl(t, v, e) -> (tab lvl) ^ (str_of_expr (Assign(v, e)))

let str_of_formal = function
  | Formal(_, v) -> v

let str_of_fdecl fdecl lvl =
  (tab lvl) ^ "def " ^ fdecl.fname ^ "(self" ^
  (if fdecl.formals = [] then
      "" else
      ", " ^ (String.concat ", " (List.map str_of_formal fdecl.formals)))
  ^ "):\n"
  ^ (String.concat "\n" (List.map (fun x-> str_of_vdecl x (lvl+1)) fdecl.locals))
  ^ "\n" ^ (tab (lvl+1)) ^
    (let l = "\n"^(tab (lvl+1)) in
    (String.concat l (List.map (fun x-> str_of_stmt x (lvl+1)) fdecl.body)))
```

```
let str_of_compute ndecl =
  str_of_fdecl ({ return_type = VoidType; fname = "compute"; formals = ndecl.args; locals =
ndecl.local_vars; body = ndecl.compute }) 1

let str_of_node ndecl =
"class " ^ ndecl.nname ^ "(): # node name\n"
 ^ (String.concat "\n" (List.map (fun x-> str_of_vdecl x 1) ndecl.local_vars)) ^ "\n"
  ^ (String.concat "\n" (List.map (fun x-> str_of_fdecl x 1) ndecl.functions)) ^ "\n\n"
^ (str_of_compute ndecl)

let node_key_value ndecl =
  "'" ^ ndecl.nname ^ "':" ^ ndecl.nname ^ "()"

let str_of_program nodes =
let res =
"def forward(node, args):
\t nodes[node].compute(*args)

def main():
\t global nodes
\t nodes = { "
^
(String.concat ", " (List.map node_key_value nodes))
^" }
# kick off the calculation by calling all start nodes' compute functions.
# using one start node for now...
\t nodes['start'].compute()

"
^
(String.concat "\n\n" (List.map str_of_node nodes))
^
"

main()
" in
(*print_endline res;*) res
```

ncml.ml:

```
open Printf
open Ast
open Sast
open Semantic
open Generator

(* file: main.ml *)
 let main () =
  try
    let source = open_in Sys.argv.(2) in
    let lexbuf = Lexing.from_channel source in
    let result = Parser.program Scanner.token lexbuf in
    (*slip the resulting string representation of the ast into a python code file that will
execute it using
```

```
     the python interpreter*)
    let output = "from ast import *\n#from DEBUG_AST_PRETTY_PRINTER import *\naststr=\"" ^
(string_of_program result) ^ "\"\nast=eval(aststr)\nast=fix_missing_locations(ast)\n#print
dump(ast)\nobj=compile(ast,\"\",\"exec\")\nexec obj" in
    let dest = open_out "IR.py" in
    output_string dest output;
    close_out dest;
    let compile_result = Sys.command "python IR.py" in
    match compile_result with
       0 -> "success!"
     | _ -> "error python ast compilation.";
  with Scanner.Eof -> "end of file input."


let main_gen () =
  try
    let source = open_in Sys.argv.(1) in
    let lexbuf = Lexing.from_channel source in
    let result = Parser.program Scanner.token lexbuf in
       let _ = Semantic.check(result) in
    let python_source_code = (str_of_program result) in
    let dest = open_out "IR.py" in
    output_string dest python_source_code;
    close_out dest;
    let compile_result = Sys.command "python IR.py" in
    match compile_result with
       0 -> "success!"
     | _ -> "error in python code compilation";
  with Scanner.Eof -> "end of file input."


let astgen = ref false
let usage = "usage: " ^ Sys.argv.(0) ^ " [-ast] <ncml-file>"

let speclist = [ ("--ast", Arg.Unit (fun () -> astgen := true), ": use python ast code
generation"); ]
let args = ref []

let _ =
  Arg.parse
    speclist
    (fun x -> args:= !args@[x] )
    usage;
  if !astgen then
    main()
  else
    main_gen()
```

parser.mly:

```
%{
       open Printf

       open Ast
```

```
       let parse_error e = print_endline e; flush stdout
%}

%token SEMI COMMA PERIOD
%token ASSIGN
%right ASSIGN
%token LBRACK RBRACK LPAREN RPAREN LBRACE RBRACE
%token <string> ID

%token INTERFACE
%token NODE
%token FUN
%token INT
%token CHAR
%token FLOAT
%token DOUBLE /* should handle this differently, or remove doubles */
%token STRING
%token BOOL
%token VOID /* an indicator, can't actually have a void "thing" */

%nonassoc ARRAY

/*TODO: rename LCONST --> ICONST, DCONST --> FCONST*/
%token <int> LCONST
%token <float> DCONST
%token <char> CCONST
%token <string> SCONST
%token <bool> BCONST

%token IF ELSE
%token BREAK
%token CONTINUE
%token FOR
%token RETURN
%token WHILE

%nonassoc NOELSE
%nonassoc ELSE

%token PRINT

%token PLUS MINUS TIMES DIVIDE MOD
%left PLUS MINUS TIMES DIVIDE MOD
%token PLUSEQ MINUSEQ TIMESEQ DIVEQ
%right PLUSEQ MINUSEQ TIMESEQ DIVEQ
%token EQ NEQ LT GT LEQ GEQ
%left EQ NEQ LT GT LEQ GEQ
%token AND OR NOT
%left AND OR NOT
%token QUOTE DQUOTE

%token FORWARD
```

```
%token TO

%token EOF

%start program
%type <Ast.program> program

%%

program:
          | /*empty*/    { [] }
/*          | program  { [] }*/
         | program node { $2 :: $1 }
/*       | program error { print_string("error found...");[] } */

node: /*think of a better way to handle these 3 optional things in a row...*/
        NODE ID LPAREN formal_list_opt RPAREN
                LBRACE  var_decl_list
                       fun_decl_list
                        compound_statement RBRACE
        { { nname       = $2;
            args        = $4;
            local_vars  = List.rev $7;
            functions   = List.rev $8;
            compute     = List.rev $9 } }

fun_decl_list :
          | /* empty */                  { [] }
          | fun_decl_list fun_decl       { $2 :: $1 }

fun_decl:
        FUN dtype ID LPAREN formal_list_opt RPAREN
           LBRACE var_decl_list
                  compound_statement RBRACE
        { { return_type = $2;
            fname       = $3;
            formals     = $5;
            locals      = List.rev $8;
            body        = List.rev $9 } }

compound_statement:
           /*nothing*/          { [] }
         | compound_statement stmt { $2 :: $1 }

stmt:
            expr SEMI                           { Expr($1) }
          | LBRACE compound_statement RBRACE  { Block($2) }
          | IF LPAREN expr RPAREN stmt %prec NOELSE { (*print_endline("IF STATMENT HIT
ELSELESS"); flush stdout; *) If($3, $5, Nostmt) }
          | IF LPAREN expr RPAREN stmt ELSE stmt   { (*print_endline("IF ELSE HIT"); flush
stdout;*) If($3, $5, $7) }
          | WHILE LPAREN expr RPAREN stmt         { While($3, $5) }
          | FOR LPAREN expr SEMI expr SEMI expr RPAREN stmt  { For($3, $5, $7, $9) }
```

```
          | BREAK SEMI    { Break }
          | CONTINUE SEMI { Continue }
          | RETURN expr SEMI      { Return($2) }
          | RETURN SEMI           { Return(Noexpr) }
          | PRINT expr SEMI     { Print($2) }
          | FORWARD actual_list TO ID SEMI { Forward($2, $4) }

formal_list_opt:
        /*empty*/               { [] }
          | formal_list         { List.rev $1 }

formal_list:
          dtype ID                    { [Formal($1, $2)] }
          | formal_list COMMA dtype ID  { Formal($3, $4) :: $1 }

actual_list_opt:
           /*empty*/            { [] }
          | actual_list         { List.rev $1 }

actual_list:
           expr             { [$1] }
          | actual_list COMMA expr { $3 :: $1 }

var_decl_list:
          /* empty */               { [] }
          | var_decl_list var_decl { $2 :: $1 }

var_decl:
           dtype ID ASSIGN expr SEMI      { VarDecl($1, $2, $4) }

expr:
          ID                  { Id($1) }
        | ID ASSIGN expr        { Assign($1, $3) }
        | CCONST              { CharLiteral($1) }
        | SCONST                { StringLiteral($1) }
        | LCONST              { IntLiteral($1) }
        | DCONST              { FloatLiteral($1) }
        | BCONST              { BoolLiteral($1) }
        | expr PLUS expr      { Binop($1, Add, $3) }
        | expr MINUS expr     { Binop($1, Sub, $3) }
        | expr TIMES expr     { Binop($1, Mult, $3) }
        | expr DIVIDE expr    { Binop($1, Div, $3) }
        | expr MOD expr             { Binop($1, Mod, $3) }
/*      | ID PLUSEQ expr      { Binop($1, Pluseq, $3) } */
        | expr EQ expr        { Binop($1, Eq, $3) }
        | expr NEQ expr             { Binop($1, Neq, $3) }
        | expr LT expr        { Binop($1, Lt, $3) }
        | expr GT expr        { Binop($1, Gt, $3) }
        | expr LEQ expr             { Binop($1, Leq, $3) }
        | expr GEQ expr             { Binop($1, Geq, $3) }
        | expr AND expr             { Binop($1, And, $3) }
        | expr OR expr        { Binop($1, Or, $3) }
        | ID LPAREN actual_list_opt RPAREN { Call($1, $3) }
```

```
        | LPAREN expr RPAREN      { $2 }
        | NOT expr              { Unop(Not, $2) }

dtype:
          CHAR   { CharType }
        | STRING { StringType }
        | INT    { IntType }
        | FLOAT  { FloatType }
        | BOOL   { BoolType }
        | VOID   { VoidType }
```

semantic.ml:

```
open Ast
open Sast

type symbol_table = {
        vars : Sast.var_decl list;
        funcs : Sast.func_decl list;
}

type env = {
        scope : symbol_table;
        nodes : node_decl list;
        curr_node : Sast.node_decl;
        in_func : bool;
        curr_ret_type : Ast.dtype;
}

let rec find_var env name =
        try
                List.find (fun (var_name, _) -> var_name = name) env.scope.vars
        with Not_found ->
                raise (Failure("Variable " ^ name ^ " not in scope"))

let rec find_func env name =
        try
                List.find (fun f -> f.func_name = name) env.curr_node.helper_funcs
        with Not_found ->
                raise (Failure("function " ^ name ^ " not defined"))

let rec find_node env name =
        try
                List.find (fun node -> node.node_name = name) env.nodes
        with Not_found ->
                raise (Failure("node " ^ name ^ " not defined"))

let same_type t arg =
        let (_, arg_type) = arg in
                if t = arg_type then
                        arg
                else
                        raise (Failure("mismatched types"))
```

```
let rec expr env = function
        | Ast.Noexpr -> Sast.Noexpr, VoidType
        | Ast.CharLiteral(lit) -> Sast.CharLit(lit), CharType
        | Ast.StringLiteral(lit) -> Sast.StringLit(lit), StringType
        | Ast.IntLiteral(lit) -> Sast.IntLit(lit), IntType
        | Ast.FloatLiteral(lit) -> Sast.FloatLit(lit), FloatType
        | Ast.BoolLiteral(lit) -> Sast.BoolLit(lit), BoolType
        | Ast.Id(id) ->
                let vdecl =
                        try
                                find_var env id
                        with Not_found ->
                                        raise (Failure ("undeclared identifier: " ^ id))
                in
                        let (name, typ) = vdecl in
                                Sast.Id(name), typ
        | Ast.Binop (e1, op, e2) ->
                let e1 = expr env e1
                and e2 = expr env e2 in
                        let (_, t1) = e1
                        and (_, t2) = e2 in
                                let t =
                                        if (not (t1 = t2)) then
                                                raise (Failure ("type mismatch"))
                                        else match op with
                                                | Ast.Add -> (match t1 with
                                                        | IntType | FloatType | StringType -> t1
                                                        | _ -> raise (Failure ("Type failure")))
                                                | Ast.Sub | Ast.Mult | Ast.Div | Ast.Mod -> (match
t1 with
                                                        | IntType | FloatType -> t1
                                                        | _ -> raise (Failure ("Type failure")))
                                                | Ast.Eq | Ast.Neq -> (match t1 with
                                                        | IntType | FloatType | CharType
                                                        | StringType | BoolType -> BoolType
                                                        | _ -> raise (Failure ("Type failure")))
                                                | Ast.Lt | Ast.Gt | Ast.Leq | Ast.Geq -> (match t1
with
                                                        | IntType | FloatType | CharType |

StringType -> BoolType
                                                        | _ -> raise (Failure ("Type failure")))
                                                | Ast.And | Ast.Or -> (match t1 with
                                                        | BoolType -> BoolType
                                                        | _ -> raise (Failure ("Mismatched types")))
                                in Sast.Binop(e1, op, e2), t
        | Ast.Unop (op, e) ->
                let e = expr env e in
                        let (_, t) = e in
                                let t = match op with
                                        | Ast.Neg -> (match t with
                                                | IntType | FloatType -> t
                                                | _ -> raise (Failure ("Type failure")))
                                        | Ast.Not -> (match t with
```

```
                                        | BoolType -> BoolType
                                        | _ -> raise (Failure ("Type failure")))
                        in Sast.Unop(op, e), t
        | Ast.Assign (id, e) ->
                        let id = find_var env id
                        and e = expr env e in
                                let (name, t1) = id
                                and (_, t2) = e in
                                        if (t1 = t2) then
                                                Sast.Assign(name, e), t1
                                        else
                                                raise (Failure ("Type mismatch in assignment"))
        | Ast.Call (func_name, params) ->
                        let args = List.map (fun s -> expr env s) params in
                                let fdecl = find_func env func_name in
                                        let types = List.rev (List.map (fun (_, typ) -> typ)
(List.rev fdecl.fparams)) in
                                                try
                                                        Sast.Call(fdecl.func_name, List.map2
same_type types args), fdecl.ret_type
                                                with Invalid_argument(x) ->
                                                        raise (Failure("Invalid number of
arguments"))

let rec stmt env = function
        | Ast.Block(s1) ->
                        let s1 = List.map (fun s -> stmt env s) s1 in
                                Sast.Block(s1)
        | Ast.Expr(e) -> Sast.Expr(expr env e)
        | Ast.Return(e) ->
                if (env.in_func = false) then
                        raise (Failure("Return statment found in illegal context"))
                else
                        let (e, typ) = expr env e in
                                if (typ = env.curr_ret_type) then
                                        Sast.Return(e, typ)
                                else
                                        raise (Failure("Return expression does not match return
type of function"))
        | Ast.If(e, s1, s2) ->
                let e = expr env e in
                        if ((snd e) = BoolType) then
                                Sast.If(e, stmt env s1, stmt env s2)
                        else
                                raise (Failure ("If condition must evaluate to a boolean"))
        | Ast.For (e1, e2, e3, s) ->
                let e2 = expr env e2 in
                        if ((snd e2) = BoolType) then
                                Sast.For (expr env e1, e2, expr env e3, stmt env s)
                        else
                                raise (Failure ("Continuation condition in For loop must be
boolean"))
        | Ast.While (e, s) ->
```

```
                let e = expr env e in
                        if ((snd e) = BoolType) then
                                Sast.While (e, stmt env s)
                        else
                                raise (Failure ("While condition must be boolean"))
        | Ast.Print (e) ->
                let e = expr env e
                in Sast.Print(e)
        | Ast.Continue -> Sast.Continue
        | Ast.Break -> Sast.Break
        | Ast.Nostmt -> Sast.Nostmt
        | Ast.Forward(expr_list, name) ->
                let args = List.map (fun p -> expr env p) expr_list in
                        let forward_node = find_node env name in
                                let types = List.rev (List.map (fun (_, typ) -> typ) (List.rev
forward_node.nparams)) in
                                        try
                                                Sast.Forward(forward_node.node_name, List.map2
same_type types args)
                                        with Invalid_argument(x) ->
                                                raise (Failure("Invalid number of arguments in
forward"))

let node_exists nodes name =
        List.exists (fun n -> n.node_name = name) nodes

let add_param env node param =
                        let Ast.Formal(typ, name) = param in
                                let exists = List.exists (fun id -> (fst id) = name)
node.nparams in
                                        if (exists) then
                                                raise (Failure("Redefinition of identifier in node
parameters"))
                                        else
                                                let params = (name, typ) :: node.nparams
                                                in {
                                                        node with nparams = params
                                                }

let add_local env node var =
        let Ast.VarDecl(typ, name, e) = var in
                let exists = List.exists (fun v -> fst v = name) node.nlocals in
                        if (exists) then
                                raise (Failure ("Redefinition of variable " ^ name ^ " in node "
^ node.node_name))
                        else
                                let scope_vars = node.nparams @ node.nlocals in
                                        let scope = {
                                                env.scope with vars = scope_vars
                                        } in
                                                let env = {
                                                        env with scope = scope
                                                } in
```

```
                                                let (_, t) = expr env e in
                                                        if (t != typ) then
                                                                raise (Failure ("type error
when initializing id " ^ name))
                                                        else
                                                                let var_list = (name, typ)
:: node.nlocals
                                                                in {
                                                                        node with nlocals =
var_list
                                                                }

let add_compute env node s =
        let stmts = s :: node.unchecked_body
        in {
                node with unchecked_body = stmts
        }

let add_func_param func param =
        let Ast.Formal(typ, name) = param in
                let exists = List.exists (fun p -> name = fst p) func.fparams in
                        if (exists) then
                                raise (Failure("Redefinition of identifier in function
parameters"))
                        else
                                let params = (name, typ) :: func.fparams
                                in {
                                        func with fparams = params
                                }


let add_func_local env node func var =
        let Ast.VarDecl(typ, name, e) = var in
                let exists = List.exists (fun v -> name = fst v) func.flocals in
                        if (exists) then
                                raise (Failure("variable redefinition"))
                        else
                                let scope = {
                                        env.scope with vars = func.fparams @ func.flocals;
                                } in
                                        let env = {
                                                env with scope = scope
                                        } in
                                                let (_, t) = expr env e in
                                                        if (t != typ) then
                                                                raise (Failure ("type error when
initializing id " ^ name))
                                                        else
                                                                let var_list = (name, typ) ::
func.flocals
                                                                in {
                                                                        func with flocals = var_list
                                                                }
```

```
let add_func_body func s =
	let stmts = s :: func.unchecked_fbody
	in {
		func with unchecked_fbody = stmts
	}

let add_func env node f =
	let exists = List.exists (fun helpf -> f.fname = helpf.func_name) node.helper_funcs in
		if (exists) then
			raise (Failure("A function with name " ^ f.fname ^ " already exists in
node " ^ node.node_name))
		else
			let func = {
				ret_type = f.return_type;
				func_name = f.fname;
				fparams = [];
				flocals = [];
				fbody = [];
				unchecked_fbody = [];
			} in
				let func = List.fold_left add_func_param func f.formals in
					let func = List.fold_left (add_func_local env node) func
f.locals in
						let func = List.fold_left add_func_body func
f.body in
							let funcs = func :: node.helper_funcs
							in {
								node with helper_funcs = funcs
							}

let add_node env node =
	if (node_exists env.nodes node.nname) then
		raise (Failure ("Node with name " ^ node.nname ^ " already exists"))
	else
		let new_node = {
			node_name = node.nname;
			nparams = [];
			nlocals = [];
			nbody = [];
			unchecked_body = [];
			helper_funcs = [];
		} in
			let new_node = List.fold_left (add_param env) new_node (List.rev
node.args) in
				let new_node = List.fold_left (add_local env) new_node
(node.local_vars) in
					let new_node = List.fold_left (add_compute env) new_node
(List.rev node.compute) in
						let new_node = List.fold_left (add_func env)
new_node (List.rev node.functions) in
							let new_nodes = (new_node :: env.nodes)
							in {
```

```
                                                        env with nodes = new_nodes
                                                }

let check_func_body env node func s =
        let scope = {
                vars = func.fparams @ func.flocals;
                funcs = node.helper_funcs;
        } in
                let env = {
                        env with scope = scope
                } in
                        let s = stmt env s in
                                let stmts = s :: func.fbody
                                in {
                                        func with fbody = stmts
                                }

let check_func env node func =
        let new_func = {
                ret_type = func.ret_type;
                func_name = func.func_name;
                fparams = func.fparams;
                flocals = func.flocals;
                fbody = [];
                unchecked_fbody = func.unchecked_fbody;
        } in
                let env = {
                        scope = env.scope;
                        nodes = env.nodes;
                        curr_node = env.curr_node;
                        in_func = true;
                        curr_ret_type = new_func.ret_type;
                } in
                        let func = List.fold_left (check_func_body env node) new_func
func.unchecked_fbody in
                                let func_list = func :: node.helper_funcs
                                in {
                                        node with helper_funcs = func_list
                                }

let check_compute env node s =
        let scope = {
                vars = node.nparams @ node.nlocals;
                funcs = node.helper_funcs;
        } in
                let env = {
                        env with scope = scope
                } in
                        let s = stmt env s in
                                let stmts = s :: node.nbody in
                                        let node = {
                                                node with nbody = stmts
                                        } in
```

```
                                        List.fold_left (check_func env) node
node.helper_funcs

let check_node env node =
        let new_node = {
                node_name = node.node_name;
                nparams = node.nparams;
                nlocals = node.nlocals;
                nbody = [];
                unchecked_body = node.unchecked_body;
                helper_funcs = node.helper_funcs;
        } in
                let env = {
                        env with curr_node = node
                } in
                        let node = List.fold_left (check_compute env) new_node
node.unchecked_body in
                                let node_list = node :: env.nodes
                                in {
                                        env with nodes = node_list
                                }

let check_start_node node_list =
        try
                let _ = List.find (fun n -> n.node_name = "start") node_list
                        in true
                        (* Do we want to check params here? *)
        with Not_found ->
                false

let check program =
        let empty_scope = {
                vars = [];
                funcs = [];
        }
        and empty_node = {
                node_name = "";
                nparams = [];
                nlocals = [];
                nbody = [];
                unchecked_body = [];
                helper_funcs = [];
        } in
                let global_env = {
                        scope = empty_scope;
                        nodes = [];
                        curr_node = empty_node;
                        in_func = false;
                        curr_ret_type = VoidType;
                } in
                        let global_env = List.fold_left add_node global_env program in
                                if (check_start_node global_env.nodes) then
                                        List.fold_left check_node global_env global_env.nodes
```

```
else
        raise (Failure ("No start node found"))
```