Nifty50

Concise Programming Language

A M Sarwar Jahan
aj2599@columbia.edu
CVN Student – COMS W4115
Fall 2014

Table of Contents

Nifty50

I would have written a shorter program but I did not have time. – A programmer

1. Motivation:

Anyone who has ever read a computer program written by someone else has one complaint in common – the code is too long. With increasing code space available to programmers on target devices, the emphasis is no longer on writing succinct lines of code that does more. This has caused the source codes to become less modular, longer, and harder to debug.

Nifty50 is a high level programming language that allocates exactly fifty lines of instruction for each module in a program. The language will be object oriented where each module is considered as an object and every source code will have a single, unique, top module (think of it as the main function in C). The objects, or modules, will take arguments and return a single value or no value. The scope of the variables can be set to global or local to the module. But does this mean Nifty50 is just another version of a regular programming language that forces the program to be in sets of 50 or less lines?

Not really. The goal of this language is to increase readability, lessen debugging time and the time needed to convert pseudo-code to actual working source code. It is a language where engineering managers and directors (years away from their programming days) will look at the top module and understand just how the program works. The language will optimize algorithms like sorting, searching, array manipulation into easy to read (and implement) function-like structures.

The language attempts to do the following:

- Provide conciseness in the way programmer write code.
- Be a programming language of choice for simple embedded systems.
- Provide ease of use in terms of using common algorithms.
- Reduce the time between developing pseudocode and actual program code.

2. Common Algorithms in Nifty50:

a. Loops: these are one of the most frequently used constructs in a source code. A for loop in C has the following syntax:

```
for ( variable initialization; condition; variable
update ) {
  Code to execute while the condition is true
}
```

In the proposed language, the conditions of the for loop are more effectively expressed as:

```
For  (Condition, {Code to execute if Condition is
true});
```

The variable is also initialized in the condition section and the variable update is done in the statement to be executed rather than as part of the loop syntax. Also, the statement can execute a module.

b. Array Manipulation: the language can be used to perform common array manipulations like insert, reverse, or search very easily. A reverse operation can be described as a global method inside an array object. Since the scope of the method is global it can be invoked with any array as long as it is of the correct type. An example of a reverse algorithm in C for an array named 'a' is:

```
for (c = n - 1, d = 0; c >= 0; c--, d++)
          b[d] = a[c];
for (c = 0; c < n; c++)
      a[c] = b[c];
```

In the proposed language, the list/array reverse function will be simple. See section 3.2 on how to add reverse as a method to an array object.

```
b = a.reverse; //reverse all elements of a to new
array b.
a = b;
```

c. Search: the algorithm for searching through an array or any data structure is also quite simple in the new language. An example of a search algorithm in an array containing characters is:

```
i = 0;
while (i < 'c'&& ch != a[i]) {
          i++;
  }
```

But for the new language, the search function will return a Boolean which will be 1 if search returns a result or a 0 if otherwise. The

```
bool search = a('c');//search is initialized to
     search for 'c' in a
search.query = 'c'; //search parameter has value
     'c', the query
search.index = 4;   //the query is found in index
     4.
```

3. Special Features of the language

3.1. Modules in Nifty50:

Modules are the building blocks of any program written in the language. A programmer can decide to divide a big program into modules that each performs a single function. Although each program can have multiple modules, every module should be declared in the main and contain an include statement. The program end statement must be in the top module.

3.2. Objects in Nifty50:

Since Nifty50 is an object oriented language, all variables and literal classes are objects with member functions. There are two methods associated with each object which is the address and swap methods. Additional user defined methods can be used to perform specific action on a single or all objects of the same type.

3.3. Interrupts:

Since the language is to be used for embedded systems, interrupts are an important feature. Interrupts can be easily declared in the main module or any of external module with the source specified. The source is a method defined in all interrupts that defines the signal or the expression handled by the interrpt.

4. Sample Program in Nifty50:

The programming language is designed to be used in particular for embedded systems application. For the sample program below, a 8 bit microcontroller is used for an occupancy sensor which detects movement in a certain peripheral area by using a PIR sensor and turns on a load (light bulb) if movement is detected. Also, the microcontroller has an ambient light sensor which it continually monitors and if there is enough ambient light, the load would not turn on for a movement. The sensor has some user inputs such as light sensitivity and light brightness that they can set externally using physical settings such as a knob.

```
//Sample program for an occupancy sensor
//Line
1.   main() {
2.   int module calcAmbient (int ambience), calcBright
     (int level);
3.   bool module calcMovement (int PortC, int
     sensitivity); //takes two arguments, the values
     read in from the PIR sensor is at Port C of
     microcontroller and sensitivity is set by the
     module calcAmbient)
```

```
4.   sensitivity = calcAmbient ( PortB); //PortB reads
     user set ambient light sensitivity
5.   turnOn: PortD.1= 1  //PortD pin 1 controls the
     load
6.   turnoff: PortD.1 = 0
7.   level = calcBright (PortA); //PortA reads user
     set brightness
8.   if (calcMovement, turnOn, turnoff); }//the if
     statement evaluates the calcMovement which is the
     condition and turns on the load if it is true or
     turns off if the condition (movement) is false.
```

5.Conclusion:

It seems that the sample program is short enough to be taken as a pseudo-code implementation. The primary goal of the new language is to reduce the time a programmer takes to write the top module that will form the skeleton of the main program and then work on the little details in the individual modules. It is imperative that the new language is easy to read and debug. The compiler for it will evaluate the syntax and semantics of each module that will aid in debugging the software as each module can be debugged independently. The dot (.) operator can be used to invoke other modules and used to perform frequently used algorithms. Overall, Nifty50 will be a object oriented programming language with user-defined modules of 50 lines of instruction or less.

2. Language tutorial for beginners.

Nifty50 is a concise programming language for simple embedded systems. Before you start, it is helpful to know that a single module in the program restricts the number of instructions you can use to 50 lines.

Here are the steps required to begin programming in the new language.

- Download the source files for the project from the class website. Available in COMS W4115 Fall 2014 class website.
- Unzip the tar file.
- Open the terminal window.
- Use the command cd <path to the project files>
- Use the command make, this will compile the language and generate an executable.
- Compile a source file to be used with the language. Name it with .mc extension.
- The executable will be name Nifty50. Type ./Nifty50 <path to source.mc>
- The terminal will execute your new language and show any outputs.

Language Reference Manual

1. Nifty50 – The Core Concept:

Nifty50 is a concise programming language for embedded system applications that is designed to present a computer program in blocks of 50 lines or less. The concept behind the restriction is to divide the program into multiple modules with a unique top level module that presents an overview of the program. Nifty50 is targeted mainly for embedded systems development on a microcontroller. The programming tenets of the language can be summarized below:

1.1. Object-oriented: Nifty50 is an object oriented programming language where types are modeled as objects that include primitives. Basic types such as integers, characters, strings, and arrays can also be treated as objects.

1.2. Strong I/O support: Since, the language is designed to be used in embedded systems, Nifty50 supports convenient input scanning and output methods.

1.3. Versatile: The language allows a programmer to place custom algorithms in the program to perform computations in a powerful way.

1.4. Compact: Each module is restricted to 50 instructions but the program is not divided into modules because of increased length but for different functionality. Each module performs specific tasks or algorithms and is connected to the top level module.

The motivation of the new language is to reduce the time a programmer takes to write the program that will form the skeleton of the main program and then work on the little details in the individual modules. It is imperative that the new language is easy to read and debug. The compiler for it will evaluate the syntax and semantics of each module that will aid in debugging the software as each module can be debugged independently.

2. Lexical Conventions:

2.1. Whitespace: The language will convert all space, new line, form feed, carriage return, and tab characters as whitespace. At least one of these characters is required to separate tokens, constants, and identifiers.

2.2 Comments: Nifty50 uses C-style comments. The language describes the comments within the /* and */ characters. Anything in between these set of characters is ignored by the compiler. The comments do not count as part of the 50 lines for a module. Comments can be single line or multi line.

```
/* This is a comment */
```

2.3. Identifiers: Identifiers are used for the distinction of variables, methods and types. An identifier is a sequence of alphanumeric characters, uppercase and lowercase, and underscores. For Nifty50, the identifiers must start with a letter and cannot start with a numeric or a special character. An identifier is case sensitive and characters like underscore (_) count as alphabetic.

2.3. Keywords: Keywords cannot be used as identifiers and are reserved for use. Keywords are special set of strings that perform a specific function in the language. The language has the following keywords:

return	if	else	for	while	int
double	string	void	port	Source	Map
Interrupt	Enum	Address	Swap	true	false
type	and	or	xor	nand	not
include	as				

2.4. Operators: The operators in this language are listed below. It has a mix of unary and binary operators.

+	-	*	/	=	==
!=	<	<=	>	>=	<<
>>	and	or	xor	nand	nor
not					

2.5. Literal Classes: Literal classes are the fundamental units of the program. A literal class is a value that may be expressed in code without the use of a new keyword.

2.5.1. Integer Literals: An integer is a sequence of digits only. Examples of integers are:-

2.5.2. Float Literals: A float literal is a sequence of digits and exactly one decimal point/period. It contains an integer or a fraction part or both seperated by a decimal '.' point. It can also be expressed as an exponent 'e' which can be signed or unsigned. There must be at least one digit before the decimal point and at least one digit after the decimal point.

2.5.3. Boolean Literals: A boolean literal can be expressed as true or false and also as its binary form.

2.5.4. String Literals: A string literal consists of a single character or a sequence of characters in double quotes. A string literal can have space, new line escape

sequence within it but cannot have a new line, form feed, or vertical tab within it. The addition "+" operator can be used to concatenate two string literals.

2.6. Seperators: The seperators used in this language delineates various aspects of program organization. Each statement is terminated with the ';' seperator. Function arguments are declared within '( )'. Arrays indexes are declared within '[ ]'. The brace characters '{ }'are used to enclose groups of statements in a module only.

3. Semantics

3.1. Types and Variables

The variables in Nifty50 are always declared with a type and identifier. The types are static and will be known at compile time. The variable holds a reference to an instance of the declared type. Since, Nifty50 is an object oriented language, all variables are objects.

3.1.1 Object types: An Object type is the most fundamental type in the language and all other types can be derived from the Object type. An Object type shall have an address associated with it. An Object type can have variables declared inside and have methods to perform specific functions. To access an object's method, the method shall be prefixed with the Object name followed by '.'. It is important to note that all objects have global scope but the methods declared inside can have local or global scopes. All objects have two inherent methods that can be called anytime – Address and Swap.

3.1.2. Array types: Arrays are 1-dimensional representation of a collection of objects of the same type. Arrays are always fixed size, indexable, and mutable. Arrays shall allow access to an item via an index, represented by an unsigned integer expression enclosed by '[' and ']'. Arrays are only available for the fundamental types.

3.1.3. Fundamental Type: The four fundamental types are Int (signed 32 bit integer), Double (64 bit floating point number), String (A string of characters), Port (One byte of data, 8 bits unsigned).

3.1.4. New types: Nifty50 shall allow new types to be declared and created. The new types shall inherit from the Object type, thus each new type shall have the Address and Swap method. Types shall contain variables and function definitions. The declaration of a type is done by 'type' keyword.

3.3. Expressions:

An expression (expr) shall be composed of identifiers, string literals, constants and can be enclosed in parentheses. The expressions in the language can be an integer expression, floating-point expression, variable-name-expression (contains a variable identifier), string expression, variable-integer expression (contains a variable and integer constants).

3.3.1. Function Calls: A function call is a postfix expression that consists of variable expression, and a list of arguments enclosed in parenthesis. The return keyword follows the argument list and indicates the type or name of the variable that the function returns.

3.3.2. The expression rules of this language are given next. Note that the precedence of the expression is followed by the order of the table.

Operator	Function	Expression Rule
()	Enclose expression	(expr, expr, …)
!	Not, check for false statement	!expr
/	Division	expr / expr
*	Multiplication	expr * expr
+	Addition	expr + expr
-	Subtraction	expr - expr
<<	Bit shift left	expr << 1 or 0
>>	Bit shift right	expr >> 1 or 0
>	Greater than	expr > expr
<	Less than	expr < expr
>=	Greater than or equal to	expr >= expr
<=	Less than or equal to	expr <= expr
!=	Not equal to	expr != expr
|	Or	expr | expr
==	Equal to	expr == expr
=	Assignment	expr = expr

3.5. Statements

Statements are sequences of code that can be executed to perform a specific function. There can be four types of statements: expression, compound, if, and iteration.

3.5.1. Expression Statements: An expression statement shall be an expression terminated by the ';' symbol. An expression symbol can be empty.

3.5.2. Compound Statements: A compound statement can be multiple statements enclosed by '{' and '}'. It can contain declarations and statements. A compound statement can be a function call with an argument list and return type.

3.5.3. If statement: If statement is a conditional statement that allows simple flow control. The expression enclosed in a parentheses is evaluated. In the case, the expression returns 1 or is true, the first statement is executed. Otherwise the second statement is evaluated. There is no need of an else keyword and the first and the second statement can either by empty but both the statements cannot be empty at any time.

```
if (expr, statement1, statement2);
```

3.5.4. Iteration Statements: These statements consist of the for and while loop statements. In the for statement, there can be three expressions, the first expression shall be evaluated once initially, the second expression shall be repeatedly evaluated as long as third expression expression is true. The while statement evaluates one or more expressions enclosed in parenthesis and executes the statements that follow as long as the expression is true.

```
For ( expr1, expr2, expr3) {expr};

While (expr) {expr};
```

3.6. Methods

Methods are code that can be used inside an object. These can be function definitions that can be invoked using the '.' operator. If a method is declared to be global, it can be invoked with any object using the '.' operator. If the a method is declared to be local, then the method can only be invoked with its associated operator.

4. Syntax

The syntax for this language can be defined with essential and optional elements. For the following definitions any expression enclosed within '[ ]' are optional elements. The words in bold are essential parts of the syntax and has to be included.

4.1 Declarations

Declarations are used within function definitions to specify the interpretation which Nifty50 gives to each identifier; they do not necessarily reserve storage associated with the identifier.

4.1.1 Variable

A variable is assigned a type and a value in the same line

```
variable_type identifier = expression
```

4.1.2. Array

An array is assigned type, identifier and expression within the same line

```
array_type identifier[index number] = expression
```

4.1.3. Method

All method objects are globally defined and are declared as

```
[global] method method_name (argument list) {statement};
```

The global key-word is optional and will allow the scope of the method to be invoked with any object. If the global keyword is missing, all methods have local scopes and are bound to the object where it is declared.

4.1.4. Function Declaration

A function can be declared as

```
function_name (argument list) return ([void|type|name])
```

Functions must declare what they are returning at the time of instantiation. A function that returns nothing is indicated by using the **void** keyword. A function can only return one value.

4.1.5. Interrupt Declaration

The keyword Interrupt shall denote a function used to handle interrupts. An Interrupt shall not return anything or receive any arguments. An interrupt has a inherent method called **handle** which can be used to assign the expression or signal that the interrupt will handle. And interrupt is declared as:

```
interrupt interrupt_name() {handle = expression };
```

4.1.6. Type Declaration

The language allows declaration of a new type that composes of one or more of the fundamental types. A type declaration shall start with the keyword type followed by its name and its constituents. A type cannot be composed of another type and it does not support inheritance.

```
type type_name {type1, type2, ..};
```

4.2 Others

4.2.1. Print

The print function can be invoked as:

```
print ("example string") ;
```

The Print function will print anything within the " " to the terminal window. C-style variables can be inserted into strings by using the %i, %d, %f, etc. symbol.

5. Program

A program shall consist of two sections: a section to contain all definitions to be used in the program and a section to contain the main where the entry point of the program resides. There are no delination between the two sections but all definitions must be declared before they are used in the program.

5.1. Definition

The definition section of the program will contain all variable and object declarations.

5.2. Main

The Main function is the entry point of the program and shall contain variables, objects, and function expressions. The main program can also call functions that reside in other modules.

5.3. Modules

Since Nifty50 can only contain 50 lines of instruction per module (that includes main and definition section above), additional modules can be added in the program. These modules shall contain function definitions and statements that can be included and invoked from the main. The additional modules are used in the program by using the Include keyword.

6. Scope Rules

To summarize the scope rules in the language:

- all objects are globally scoped.
- the methods in an object can be locally or globally scoped.
- the variables in an object are always bound to the object it is in.
- variables declared in a function are locally scoped.
- all types are globally scoped regardless of which module they are declared in.

4. Project Plan

Process for Planning, Specification, Development, and Testing:

This project was carried out by a one person team and therefore proper planning was required so that all deliverables can be submitted on time. In order to complete the project, the following actions were outlined:

- Maintain a weekly deadline to complete smaller assignments in the project.
- Communicate with the professor and TAs regarding project specification.
- Use feedback from the teaching staff in the specification and development.
- Use an easy-to-use IDE (Eclipse OcalIDE) to aid in Ocaml development.
- Develop a comprehensive testing plan to test software.

Although, a lot of time was spent in the first three stages of the project, more time could have been devoted to test the language in detail especially the project could have had a greater focus on the testing for corner and exceptional cases.

Programming Style Guide:

The style guide for this project was heavily influenced from the programming style used in microC. The project was initialized by taking microC as a template and adding additional features as time progressed. Also, the OcalIDE has Ocaml syntax colorization and indentation which helped in stylizing the codes written for the project. Here are the basic style guides used:

- Comments before (almost) every block of code. Ocaml code is indecipherable in a week after being written and comments were heavily used to indicate code functionality.
- Indentation level is increased when declaring a function with at least an argument.
- When pattern matching, every case is in a separate line with the indentation aligned with the previous line.
- Function names are in lower case and names describe what they do.
- Underscores are used to separate words in names.
- A commented header for file name, author, class
- An end of file comment.

An example of the style guide is given from the Nifty50.ml file.

```
1 (* Nifty50.ml by Aamir Sarwar Jahan, COMS W4115*)
2
3 (* use the Ast.ml and Compile.ml file *)
4 type action = Ast | Compiler
5
6 let _ =
7   let action = if Array.length Sys.argv > 1 then
8     List.assoc Sys.argv.(1) [ ("-a", Ast);
9                               ("-c", Compiler)]
10  else Compiler in
11  let lexbuf = Lexing.from_channel stdin in
12  let program = Parser.program Scanner.token lexbuf in
13  match action with
14    Ast -> let listing = Ast.string_of_program program
15           in print_string listing
16  | Compiler -> Execute.execute_prog (Compiler.translate program)
17
18 (* End of File *)
```

Project Timeline:

The front-end development was mainly divided into four sections.

- Proposal [September 2nd and 3rd week]
- LRM [October 3rd and 4th week]
- Parser [October 3rd and 4th week]
- Lexer [October 4th week]

Additional sections of the project and most of the heavy lifting was done during November and December.

- Type of compiler [November 2nd week]
- Semantic Checking [November 2nd week]
- Test Hello World [November 3rd week]
- AST to C [December 1st week]
- LRM update [December 1st week]
- Test suites [December 2nd week]
- Final Report [December 2nd and 3rd week]

Software Development Environment (tools and language):

To develop this language, the following software tools were used on a Macintosh OS X 10.10:

a. OCalIDE: this is an Ocaml plug-in for Eclipse which has source editor for modules (.ml), interfaces (.mli), parsers (.mly) and lexers (.mll) files. The plug-in can be used in eclipse.

b. Ocaml version 4.01.0

c. Ocaml OPAM package: this is the open-source package manager edited by OcamlPro.

d. Git Version control: this is used as the subversion client to manage the project.

Project Log:

The project history from the Git repository is given below. Some repeated commits have been omitted for conciseness.

```
Aamir Jahan 2014-12-17 20:39:08 -0500 style fix
Aamir Jahan 2014-12-17 19:01:23 -0500 update language reference manual
Aamir Jahan 2014-12-17 08:37:28 -0500 removed dead code
Aamir Jahan 2014-12-16 21:44:26 -0500 added test
Aamir Jahan 2014-12-16 19:03:25 -0500 removed some unused functionality
Aamir Jahan 2014-12-16 19:41:52 -0500 added conditional test
Aamir Jahan 2014-12-16 18:05:20 -0500 test for include files
Aamir Jahan 2014-12-16 12:54:58 -0500 Added a couple of tests for exceptions.
Aamir Jahan 2014-12-16 12:48:23 -0500 precedence change for < >
Aamir Jahan 2014-12-16 15:47:17 -0500 Print compiler errors to stderr.
Aamir Jahan 2014-12-16 15:35:01 -0500 unclosed comment raise exception
Aamir Jahan 2014-12-16 02:51:18 -0500 cleanup
Aamir Jahan 2014-12-16 01:37:58 -0500 asm test pretty
Aamir Jahan 2014-12-16 01:33:02 -0500 small asm change
Aamir Jahan 2014-12-16 00:45:41 -0500 check for scopes.
Aamir Jahan 2014-12-15 20:09:35 -0500 sast: forgot a variable check
Aamir Jahan 2014-12-15 19:44:13 -0500 cleanup
Aamir Jahan 2014-12-15 14:14:57 -0500 removed test directory
Aamir Jahan 2014-12-15 14:06:09 -0500 more tests
Aamir Jahan 2014-12-15 12:10:25 -0500 cleanup
Aamir Jahan 2014-12-15 12:09:25 -0500 catch exception on syntax error
Aamir Jahan 2014-12-15 10:41:14 -0500 ast printer removed
Aamir Jahan 2014-12-14 12:28:08 -0500 cleanup
Aamir Jahan 2014-12-14 12:37:41 -0500 precedence test added
Aamir Jahan 2014-12-14 11:26:43 -0500 operator precedence fixed
Aamir Jahan 2014-12-14 07:13:02 -0500 sign tests added for arithmetic binops
Aamir Jahan 2014-12-13 17:33:28 -0500 cleanup
Aamir Jahan 2014-12-13 15:14:11 -0500 exception implemented
Aamir Jahan 2014-12-13 10:03:40 -0500 basic try/catch/throw implementation
Aamir Jahan 2014-12-13 09:12:36 -0500 Operators implemented and tested
Aamir Jahan 2014-12-13 07:43:56 -0500 comment scanner fixed
Aamir Jahan 2014-12-12 09:20:40 -0500 added multiline comments
Aamir Jahan 2014-12-12 00:08:43 -0500 Added test for variable declarations.
Aamir Jahan 2014-12-11 23:03:12 -0500 Implemented proper labels for loops. Added break and continue
keywords. Added tests.
Aamir Jahan 2014-12-10 22:01:00 -0500 Removed interpreter.
Aamir Jahan 2014-12-10 20:04:34 -0500 Implemented proper labels.
Aamir Jahan 2014-12-10 11:04:02 -0500 Added .gitignore.
Aamir Jahan 2014-12-09 14:21:43 -0500 removed test from make
Aamir Jahan 2014-12-02 00:35:04 -0500 new test case, cleanup
Aamir Jahan 2014-12-02 00:11:25 -0500 microc deleted
Aamir Jahan 2014-11-22 02:14:32 -0500 Added .gitignore.
Aamir Jahan 2014-11-22 02:11:46 -0500 Added microc.
Aamir Jahan 2014-11-22 02:10:36 -0500 Put some file in nifty, moved testing code to nifty
Aamir Jahan 2014-10-17 01:24:07 -0400 Give error if test file is empty given.
Aamir Jahan 2014-10-17 00:39:13 -0400 Adding test-gcc.txt.
Aamir Jahan 2014-10-17 00:24:50 -0400 Various fixes. Tester works.
Aamir Jahan 2014-10-17 00:09:09 -0400 Cleanup. Added microc as template.
```

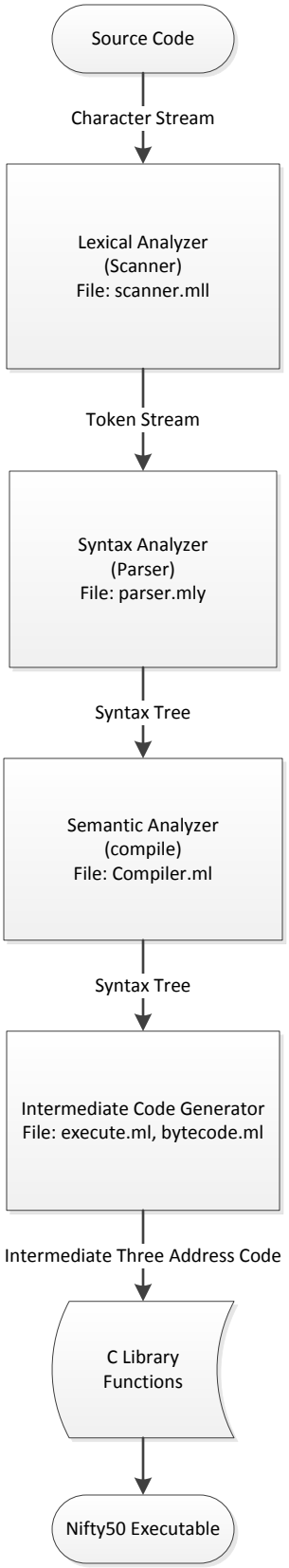
5. Architectural Design

Interface Between Components:

The translator has the functional blocks as described in the diagram in next page. At first, when a source code is fed to the scanner, lexical analysis converts the character streams into tokens as specified by the language. The parser then performs a syntax analyzer on the tokens to make sure that the source code is syntatically correct. The output of the syntax analyzer is an abstract syntax tree constructed from the grammar specified in the language.

The semantic analyzer performs a check on the syntatically correct abstract syntax tree and creates symbol table from the compiler.ml file. The intermediate code is generated by execute.ml and bytecode.ml. These are modules taken from the MicroC project that converts the Nifty50 instructions to three adresss code that performs operation on a virtual stack. Finally, a number of C functions are used to convert the intermediate representation into C code.

Nifty50 Translator: Block Diagram



6. Test Plan

Description of test cases:

The test plan to carry out experiments on the compiler was based on the following:

- Test code to carry out the basic arithmetic on the language.
- Test code to carry out basic printing on the language.
- Test code to carry out conditional statement (if) on the language.
- Test code to execute iteration statement (while and for) on the language.
- Test code to check for scoping: local and global.
- Test code to carry out a complete funciton with arguments and return value.
- Test code to check if the number of instructions is 50 or less.

As can be realized from above, only test cases to check the functional aspects of the languae was carried out. More time or better planning was required to carry out testing on little details that can induce unknown behavior or compiler error.

Automation in test procedure:

A shell script was used to test all the cases listed above and output was checked visually in an output file. The shell script was same as the one used in MicroC project.

Example test scripts:

```
/* tests math */

int main()
{
     Int example = 0;
     If (1) print(1);
  print (42 / 1 * 2 - 6 + 3);
  Return example;
}

/*tests condition*/

int main()
{
 int example;
 int a = 3;
 if (a<=3, example = 42, example 0);
 Return example;
}
/* tests iteration */
```

```
int main()
{
 int i = 0;
 int example = 0;
 int j = 0;
 int example2 = 1
 for(i=0;i<3;i +1) {example = example + 1};
 print (“example”)

 while(j<3,j + 1) {example = example + 1};
}

/* test printing */

int main()
{
  print (“Hello World”);
}
```

7. Lessons Learned

Since, I have undertaken this project as a CVN student, I am a one-man team that has avoided the usual pitfalls of working with others but at the same time I have missed out on positives that arise out of working in an effective team. Some of the key lessons that I have learnt along the way are:

- Stick strictly to the weekly plan: Since, the project was announced at the very start of the semester and I knew that I had to work on it alone, I made the plan of dividing the project into weekly assignments with short-term goals eventually working towards the completion of the project. While I did reasonably well to perform my weekly tasks for most weeks, I feel that I have not done enough on the projects on weeks which had weekly assignments due for this class or the other. By far, this has been the biggest road-block for me since I realized I have spent an entire week on a homework assignment without doing any legwork on the project. Even though significant work could not be performed, I could send emails to TAs and the professor to sort out problems from previous weeks.
- Perform the easy tasks first: At the beginning of the project, I was too concerned with carrying out the syntactic sugar in my language. I quickly realized that I should start from a more generic based approach and focus on getting the easier parts of the language first. I started using the MicroC compiler as a base and it has been easier to add features to a platform rather than starting with something completely new.
- Having a team can help: This was my first software project that I have done as a single member team. I realized that having other members certainly helps since you can get and provide assistance on topics that do not require the next level of support (TA's and professor).

My advice to future CVN students undertaking this project will be the following:

- Have a weekly plan: I had to balance my full-time work with this project as well as project from another class. The only way I could have succeeded is by breaking the big task (project) into smaller weekly deliverables.
- Get used to the development environment as soon as possible: Ocaml is a new language and getting used to compiling programs in command line or IDE (I used Eclipse) takes time. Sooner you acclimatize yourself, the more time you can spend doing actual development.
- Get as much assistance from TA and Professor: Since you will be working alone, do not hold back questions to the teaching faculty.
- Start early: this of course requires no explanation. There are steep learning curves involved and the earlier you start, the better off you will be.

8. Appendix

All source codes except:

- bytecode.ml from MicroC project
- execute.ml from MicroC project
- testall.sh from MicroC project
- test cases in /tests folder

```
1 (*Ast.ml file for Nifty50 by Aamir Jahan, COMS W4115, Fall 2014*)
2
3 (* declaring all the binary and unary operator type in Nifty50 *)
4 type binop = Add | Sub | Mult | Div | Equal | Neq | Less | Leq |
  Greater | Geq | Bsl | Bsr | And | Or
5 | Xor | Nand | Nor
6
7 type unop = Not
8
9 (* declare modules for the language *)
10 type modules = Variable | Class | Function | Interrupt
11
12 (* declare all data types *)
13 type datatypes = Array of datatypes * string | Int | Double |
  String | Port | Void |NewType of string
14
15 (* declaring type expression; an expr can be:- *)
16 (* int, binary, hexadecimal, port, string, binop, assignment, call
  *)
17 type expr =
18     ILiteral of int
19   | FLiteral of float
20   | SLiteral of string
21   | Variable of datatypes
22   | Id of string
23   | Unop of unop * expr
24   | Binop of expr * binop * expr
25   | Assign of string * expr
26   | CastType of datatypes * expr      (* check this later *)
27   | Address of string * expr
28   | GetAddress of string
29   | ArrayIndex of string * string
30   | Method of expr * expr
31   | Swap of string
32   | Signal of string * expr
33   | Map of string * string * expr * expr
34   | Call of string * expr list
35   | MethodCall of string * string * expr list
36   | Noexpr
37
38 (* statement can be a block, expression, if, for, while *)
```

```
39 type stmt =
40     Block of stmt list
41   | Expr of expr
42   | Return of expr
43   | If of expr * stmt * stmt
44   | For of expr * expr * expr * stmt list
45   | While of expr * stmt
46
47 (* declaring variable data types *)
48 type variable_declaration = {
49   vtype: datatypes;
50   vname: string;
51   }
52 (* function declarations are fname, formals, locals, body *)
53 type function_declaration = {
54   rtype : datatypes;
55   fname : string;
56   formals : variable_declaration list;
57   locals : variable_declaration list;
58   body : stmt list;
59   }
60
61 (* declaring interrupt type *)
62 type interrupt_declaration = {
63   iname : string;
64   ilocals : variable_declaration list;
65   ibody : stmt list;
66 }
67
68 type type_declaration = {
69   ytype: datatypes;
70   yname: string;
71   yproperties: variable_declaration list;
72   yfunctions: function_declaration list;
73 }
74
75 let first = fun (a,b,c,d) -> a
76 let second = fun (a,b,c,d) -> b
77 let third = fun (a,b,c,d) -> c
78 let fourth = fun (a,b,c,d) -> d
79
```

```
80
81
82 (* program returns a list of strings and func_decl list *)
83 type program = function_declaration list * type_declaration list
84                 * interrupt_declaration list * variable_declaration
   list
85
86 (* End of File*)
```

```
(* Scanner for Nifty50 by Aamir Jahan, COMS W4115, Fall 2014 *)

{ open Parser LineCount}

(* declare digits, exponent, lxms, doubles, and strings *)
let digits = ['0'-'9']+
let exp = 'e'('+'|'-')? digits
let lxm = ['a'-'z' 'A'-'Z']['a'-'z' 'A'-'Z' '0'-'9' '_']*
let double = (digits exp? | digits '.' digits? exp? | digits '.' exp)
let str = '"' [^'"']* '"'


rule token = parse
  ['\t' ' ' '\n' '\r' ] {token lexbuf} (*these tokens are considered as whitespace*)
| "/*"  {comment lexbuf}              (*denotes start of a comment*)
(* Groupings, Separators, Punctuators *)
| '('   { LPAREN }
| ')'   { RPAREN }
| '{'   { LBRACE }
| '}'   { RBRACE }
| ';'   { SEMI }
| ','   { COMMA }
| '['   { LBRACKET }
| ']'   { RBRACKET }
| '.'   { METHOD }
(* Operators, Binary and Boolean *)
| '+'   { PLUS }
| '-'   { MINUS }
| '*'   { TIMES }
| '/'   { DIVIDE }
| '='   { ASSIGN }
| "=="  { EQ }
| "!="  { NEQ }
| '<'   { LT }
| "<="  { LEQ }
| ">"   { GT }
| ">="  { GEQ }
| "<<"        { BSL }
```

```
39 | ">>"         { BSR }
40 | "and"        { AND }
41 | "or"         { OR }
42 | "xor"        { XOR }
43 | "nand"       { NAND }
44 | "nor"        { NOR }
45 | "not"        { NOT }
46 (* Conditional Statements, Loops *)
47 | "return"     { RETURN }
48 | "if"         { IF }
49 | "else"       { ELSE }
50 | "for"        { FOR }
51 | "while"      { WHILE }
52 (* data types *)
53 | "int"        { INT }
54 | "double"     { DOUBLE }
55 | "string"     { STRING }
56 | "port"       { PORT }
57 | "void"       { VOID }
58 | "True"       { TRUE }
59 | "False"      { FALSE }
60 (* Special Functions *)
61 | "Source"     { SOURCE }
62 | "Map"        { MAP }
63 | "Interrupt"  { INTERRUPT }
64 | "Enum"       { ENUM }
65 | "Address"    { ADDRESS }
66 | "Swap"       { SWAP }
67 | "Type"       { TYPE }
68 (* digits, identifiers *)
69 | digits as integer { ILITERAL(int_of_string integer) }
70 | double as dble { FLITERAL(float_of_string dble) }
71 | lxm as id    {ID(id) }
72 | str as slit {SLITERAL(slit)}
73 (* EOF functions *)
74 | eof { EOF }
75 | _ as char { raise (Failure("Illegal Character " ^ Char.escaped
   char)) }
76
77 and comment = parse
78   "*/" { token lexbuf }
```

```
79 | _    { comment lexbuf }
80
81 (* End of File *)
82
```

```
 1 (* Module to count lines in Nifty50 *)
 2 open Scanner
 3
 4 let lines = ref 0
 5
 6 rule count = parse
 7   '\n'  {incr lines; count lexbuf }
 8 {
 9 let main() =
10 let lexbuf = Lexing.from_channel stdin in
11   count lexbuf;
12   if lines > 50 then raise (Failure("More than 50 instructions"))
13 }
```

```
 1 /* Parser for Nifty50 by Aamir Jahan, COMS W4115, Fall 2014 */
 2
 3 %{ open Ast %}
 4
 5 /* tokens expressed in nifty50, same as the ones in scanner */
 6 %token SEMI LPAREN RPAREN LBRACE RBRACE LBRACKET RBRACKET COMMA
   METHOD
 7 %token PLUS MINUS TIMES DIVIDE ASSIGN
 8 %token EQ NEQ LT LEQ GT GEQ
 9 %token AND OR XOR NAND NOR NOT BSL BSR
10 %token RETURN IF ELSE FOR WHILE
11 %token <int> ILITERAL
12 %token <float> FLITERAL
13 %token <string> SLITERAL
14 %token <string> ID
15 %token EOF
16
17 /* other operators */
18 %token ADDRESS SWAP SOURCE MAP
19 %token INT DOUBLE PORT STRING
20 %token ENUM STRING
21 %token INTERRUPT TYPE METHOD VOID
22 %token TRUE FALSE
23
24 /* associativity rules, priority increases downwards */
25 %nonassoc NOELSE
26 %nonassoc ELSE
27 %right ASSIGN
28 %left NOT
29 %left EQ NEQ
30 %left ADDRESS
31 %left SWAP
32 %left COMMA
33 %left LT GT LEQ GEQ
34 %left AND OR XOR NAND NOR
35 %left BSL BSR
36 %left PLUS MINUS
37 %left TIMES DIVIDE
38
39 %start program
40 %type <Ast.program> program
```

```
41
42 %%
43
44
45 /* Grammar portion */
46
47 program:
48    /* nothing */ { [], [], [], []}
49  | program vdecl { List.rev ($2 :: first $1), second $1, third $1,
   fourth $1 }
50  | program fdecl { first $1, List.rev ($2 :: second $1), third $1,
   fourth $1 }
51  | program idecl { first $1, second $1, List.rev ($2 :: third $1),
   fourth $1 }
52  | program ydecl { first $1, second $1, third $1, List.rev ($2 ::
   fourth $1) }
53
54 vdecl:
55    return_type ID SEMI
56    {
57     {
58       vtype = $1;
59       vname = $2;
60     };
61    }
62     | return_type ID array_id SEMI
63      {
64       {
65         vtype = Array($1, $3);
66         vname = $2;
67       }
68     }
69
70 fdecl:
71    return_type ID LPAREN formals_opt RPAREN LBRACE vdecl_list
   stmt_list_opt RBRACE
72    {
73     {
74       rtype = $1;
75       fname = $2;
76       formals = $4;
```

```
77       locals = List.rev $7;
78       body = List.rev $8;
79       }
80     }
81
82 idecl:
83   INTERRUPT ID LBRACE vdecl_list stmt_list_opt RBRACE
84   {
85     {
86       iname = $2;
87       ilocals = List.rev $4;
88       ibody = List.rev $5;
89     }
90   }
91
92 ydecl:
93   TYPE ID LBRACE vdecl_list fdecl_opt RBRACE
94   {
95     {
96       ytype = NewType($2);
97       yname = $2;
98       yproperties = $4;
99       yfunctions = $5;
100    }
101  }
102
103 /*describing formals list */
104 formals_opt:
105     /* nothing */ { [] }
106   | formal_list   { List.rev $1 }
107
108 formal_list:
109   formal /* nothing */        { [$1] }
110   | formal_list COMMA formal  { $3 :: $1 }
111
112 formal:
113   return_type ID
114   {
115     {
116       vtype = $1;
117       vname = $2;
```

```
118     }
119   }
120   | return_type ID array_id
121   {
122     {
123       vtype = Array($1, $3);
124       vname = $2;
125     }
126   }
127
128 /* describing variable decl list */
129 vdecl_list:
130     /* nothing */    { [] }
131   | vdecl_list vdecl { $2 :: $1 }
132
133 /* describing array initialization*/
134 array_id:
135   LBRACKET ILITERAL RBRACKET { string_of_int $2 }
136   | LBRACKET ID RBRACKET { $2 }
137
138 array_size:
139   ILITERAL { Iliteral($1) }
140   | ID { Id($1) }
141
142 /* describing function declaration */
143 fdecl_list:
144   | fdecl       { [$1] }
145   | fdecl_list fdecl { $2 :: $1 }
146
147 fdecl_opt:
148     /* nothing */ { [] }
149   | fdecl_list   { List.rev $1 }
150
151 /*describing all possible return types */
152 return_type:
153    VOID         {Void}
154   | PORT        {Port}
155   | INT         {Int}
156   | STRING      {String}
157   | DOUBLE      {Double}
158   | ID          {NewType($1)}
```

```
159
160 /* describing statements */
161 stmt_list_opt:
162       /* nothing */  { [] }
163   | stmt_list { List.rev $1 }
164
165 stmt_list:
166    stmt /* nothing */  { [$1] }
167   | stmt_list stmt { $2 :: $1 }
168
169 stmt:
170     expr SEMI { Expr($1) }
171   | RETURN expr SEMI { Return($2) }
172   | LBRACE stmt_list RBRACE { Block(List.rev $2) }
173   | IF LPAREN expr RPAREN stmt %prec NOELSE { If($3, $5,
    Block([])) }
174   | IF LPAREN expr RPAREN stmt ELSE stmt    { If($3, $5, $7) }
175   | FOR LPAREN expr_opt SEMI expr_opt SEMI expr_opt RPAREN LBRACE
    stmt_list_opt RBRACE
176      { For($3, $5, $7, $10) }
177   | WHILE LPAREN expr RPAREN LBRACE stmt_list_opt RBRACE
    { While($3, $6) }
178
179
180 expr_opt:
181     /* nothing */ { Noexpr }
182   | expr          { $1 }
183
184 expr:
185   | ILITERAL            { Iliteral($1) }
186   | FLITERAL            { Fliteral($1) }
187   | SLITERAL            { Sliteral($1) }
188   | ID                  { Id($1)}
189   | ID array_id         {ArrayIndex($1, $2) }
190   | ID  METHOD ID array_id { METHOD(Id($1), (ArrayIndex($3,
    $4))) }
191   | NOT expr          { Unop(Not, $2) }     /*Test this line*/
192   | expr PLUS   expr  { Binop($1, Add,   $3) }
193   | expr MINUS  expr  { Binop($1, Sub,   $3) }
194   | expr TIMES  expr  { Binop($1, Mult,  $3) }
195   | expr DIVIDE expr  { Binop($1, Div,   $3) }
```

```
196   | expr EQ     expr  { Binop($1, Equal, $3) }
197   | expr NEQ    expr  { Binop($1, Neq,   $3) }
198   | expr LT     expr  { Binop($1, Less,  $3) }
199   | expr LEQ    expr  { Binop($1, Leq,   $3) }
200   | expr GT     expr  { Binop($1, Greater,  $3) }
201   | expr GEQ    expr  { Binop($1, Geq,   $3) }
202   | expr OR    expr   { Binop($1, Or,   $3) }
203   | expr AND   expr   { Binop($1, And,   $3) }
204   | expr NOR    expr  { Binop($1, Nor,   $3) }
205   | expr NAND   expr  { Binop($1, Nand,   $3) }
206   | expr BSR    expr  { Binop($1, Bsr,   $3) }
207   | expr BSL    expr  { Binop($1, Bsl,   $3) }
208   | expr XOR    expr  { Binop($1, Xor,   $3) }
209   | expr ASSIGN expr  { Assign($1, $3) }
210   | ID ADDRESS        { GetAddress($1)}
211   | ID METHOD SOURCE ASSIGN expr  { Signal($1, $5) }
212   | ID METHOD ADDRESS ASSIGN expr { Address($1, $5)}
213   | ID METHOD ADDRESS ASSIGN ID METHOD ADDRESS { Address($1,
    Id($5))}
214   | ID METHOD SWAP LPAREN RPAREN {Swap($1)}
215   | ID LPAREN actuals_opt RPAREN { Call($1, $3) }
216   | ID METHOD ID LPAREN actuals_opt RPAREN { MethodCall($1, $3,
    $5) }
217   | ID METHOD MAP LPAREN ID COMMA expr COMMA expr RPAREN { Map($1,
    $5, $7, $9)}
218   | LPAREN expr RPAREN { $2 }
219
220 actuals_opt:
221     /* nothing */ { [] }
222   | actuals_list  { List.rev $1 }
223
224 actuals_list:
225     expr                    { [$1] }
226   | actuals_list COMMA expr { $3 :: $1 }
```

```
 1 (* Compiler.ml for Nifty50 by Aamir Jahan, COMS W4115 *)
 2
 3 open Ast
 4 module StringMap = Map.Make(String)
 5
 6 (*stores global variables *)
 7 type varSymbolTableEntry = {
 8   data_type : datatypes;
 9 }
10
11 (*stores functions *)
12 type funcSymbolTableEntry = {
13   ftype : datatypes;
14   fparameters : datatypes StringMap.t;
15   flocals :  datatypes StringMap.t;
16   fbodylist : stmt list;
17 }
18
19 (*stores Interrupt handlers and local variables*)
20 type interruptSymbolTableEntry = {
21   inlocals : datatypes StringMap.t;
22   inbodylist : stmt list;
23 }
24
25 (*stores user-defined types *)
26 type typeTableEntry = {
27   thetype : datatypes;
28   properties : datatypes StringMap.t;
29   functions : funcSymbolTableEntry StringMap.t;
30 }
31
32 (*environment that describes all types*)
33 type environment = {
34   niftyVarSymbolTable : datatypes StringMap.t;
35   niftyFuncSymbolTable : funcSymbolTableEntry StringMap.t;
36   niftyInterruptSymbolTable : interruptSymbolTableEntry
   StringMap.t;
37   niftyTypesSymbolTable : typeTableEntry StringMap.t;
38 }
39
40 let checkFunctionBody theBody env construct =
```

```
41
42 let rec output_function_expr = function
43     Sliteral(l) -> l
44   | Fliteral(l) -> string_of_float l
45   | Iliteral(l) ->  string_of_int l
46   | Pliteral(l) -> l
47   | Id(s) ->  s
48   | Binop(e1, o, e2) -> "(" ^
49       string_of_expr e1 ^ " " ^
50       (match o with
51         Add -> "+"
52       | Sub -> "-" | Mult -> "*" | Div -> "/"
53       | Equal -> "==" | Neq -> "!="
54       | Less -> "<" | Leq -> "<=" | Greater -> ">" | Geq -> ">="
55       | Or -> "or"       | And -> "and"
56       | Bsr -> ">>" | Bsl -> "<<" | Xor -> "xor" ) ^ " " ^
57       string_of_expr e2 ^ ")"
58   | Unop(o, e)-> "(" ^
59     (match o with
60       Not -> "!"
61       ) ^ output_function_expr e^")"
62   | Assign(v, e) -> output_function_expr v ^ " = " ^
  output_function_expr e
63   | Call(f, el) ->
64   (match f with
65     "print" -> "printf" ^ "(" ^ String.concat ", &" (List.map
  output_function_expr el) ^ ")"
66     | _ -> f ^ "(" ^ String.concat ", &" (List.map
  output_function_expr el) ^ ")"
67   )
68   | MethodCall(v, f, el)->
69   (match f with
70       _          -> v ^ "." ^ f ^ "(" ^ String.concat ", " (List.map
  output_function_expr el) ^ ")"
71   )
72   | Noexpr -> ""
73   | CastType(a, b)->""
74   | Address(a, b)-> "Util_Move((void *)(&"^ a ^ "), (void*)(&" ^
  output_function_expr b ^ "), sizeof(" ^ a ^ "))";
75   | GetAddress(a)-> "(&" ^ a ^")"
76   | ArrayIndex(a, b)-> a ^ "[" ^  b ^ "]"
```

```
 77   | Method(a, b)-> output_function_expr a ^ "." ^
   output_function_expr b
 78   | Signal(a, b)-> "Interrupt_SetISR(" ^ a ^ "____Handler, &"^ a
   ^");\n Interrupt_SetSignal("
 79           ^ output_function_expr b ^ ", &" ^ a ^ ")"
 80   | Swap(a)-> "Util_Swap((void *)&" ^  a ^ ", sizeof(" ^  a ^ "))"
 81   | Map(a, b, c, d)->
 82       "Array_Map((void *)" ^a^", (void (*)(void *))"^b^",
   "^output_function_expr c^", " ^ output_function_expr d ^")"
 83   | Variable(a)-> "" in
 84
 85 let rec output_function_stmts = function
 86   | Block(stmts)  -> "{\n" ^ String.concat "" (List.map
   output_function_stmts stmts) ^ "}\n";
 87   | Expr(expr)    -> output_function_expr expr ^ ";\n";
 88   | Return(expr)  -> "return " ^ output_function_expr expr ^ ";\n"
 89   | If(e, s, Block([])) -> "if (" ^ output_function_expr e ^ ")\n"
   ^ output_function_stmts s;
 90   | If(e, s1, s2)       -> "if (" ^ output_function_expr e ^ ")\n"
   ^
 91       output_function_stmts s1 ^ "else\n" ^ output_function_stmts
   s2;
 92   | For(e1, e2, e3, s)  -> "for (" ^ output_function_expr e1  ^
   " ; " ^ output_function_expr e2 ^ " ; " ^
 93       string_of_expr e3  ^ ") " ^ String.concat "" (List.map
   output_function_stmts s) ;
 94   | While(e, s)         -> "while (" ^ string_of_expr e ^ ")\n{" ^
   String.concat "" (List.map output_function_stmts s) ^"}"
 95
 96
 97   in List.map output_function_stmts (List.rev theBody);;
 98
 99 (* prints the type  *)
100 let rec output_type key character = function
101   | Port                -> "unsigned char " ^ character ^ key
102   | Int                 -> "int " ^ character ^ key
103   | Double              -> "double " ^ character ^ key
104   | String              -> "char *" ^ character ^ key
105   | NewType(newtype)    -> newtype ^ character ^ " " ^ key
106   | Array(basetype, sz) -> (output_type key character basetype ) ^
   "[" ^ sz ^ "]"
```

```
107
108 let rec output_typefunc key = function
109   Void                    -> "void " ^ " " ^ key
110   | Port                  -> "unsigned char "  ^ key
111   | Int                   -> "int " ^ key
112   | Double                -> "double " ^ key
113   | String                -> "char *" ^ key
114   | NewType(newtype)      -> newtype ^ " " ^ key
115
116 let output_newtype = function
117 | NewType(newtype) -> newtype ^ " "
118
119 (* returns the global variable, AST checked post semantically*)
120  let output_globals global_table env =
121   let output_each_global key value =
122     print_endline (output_type key "" value  ^ ";") in
123   StringMap.iter output_each_global global_table
124
125  let output_params param_table =
126   let output_each_param key value =
127     print_string (output_type key "*" value ^ ", ") in
128   StringMap.iter output_each_param param_table
129
130
131 (* spits out the interrupt handlers in C form, post semantically
    checked AST *)
132 let output_interrupts interrupt_table env =
133   let output_each_interrupt key value =
134     print_endline ("Interrupt " ^ key ^ ";"); in
135     StringMap.iter output_each_interrupt interrupt_table;
136   let output_each_interruptbody key value =
137     let interruptbody =
138     "void " ^ key ^ "____Handler (int sig)\n{" in
139     print_endline (interruptbody);
140     output_globals value.inlocals env;
141     (*eventually output the body here as well*)
142     List.map print_endline (checkFunctionBody value.inbodylist env
    Interrupt);
143     (* List.map print_endline (List.map output_function_stmts
    value.inbodylist); the gangsta line*)
144     print_endline ("}\n") in
```

```
145     StringMap.iter output_each_interruptbody interrupt_table;;
146
147
148  let output_functions function_table typeFunction env =
149   let output_each_function key value =
150     print_string (output_typefunc key value.ftype ^ "(");
151     output_params value.fparameters;
152     if typeFunction != Ast.Void then print_string ((output_newtype
    typeFunction) ^" *itself");
153     print_endline (")");
154     print_endline ("{");
155     output_globals value.flocals env;
156     (*eventually output the body here as well*)
157     List.map print_endline (checkFunctionBody value.fbodylist env
    Function);
158     (* List.map print_endline (List.map output_function_stmts
    value.fbodylist); the gangsta line*)
159     print_endline ("}\n") in
160     StringMap.iter output_each_function function_table;;
161
162  let output_types types_table env =
163   let output_each_type key value =
164     print_endline ("typedef struct {");
165
166     output_globals value.properties env;
167     print_endline ("} " ^ key ^ ";\n");
168     (*eventually output the body here as well*)
169     output_functions value.functions value.thetype env in
170     StringMap.iter output_each_type types_table;;
171
172 (*utility to print out keys of each string Map*)
173 let print_vars key value =
174     print_string(key ^ " \n");
175     (*array type checking.  This is amazing!! *)
176     let thetype = Ast.Array(Int, "function") and thattype = value
    in
177     if thetype = thattype then print_endline "Yes"
178   else print_endline "No";;
179
180 let print_funcs key value =
181     print_string("Function "^ key ^": \n");
```

```
182     print_endline "Parameters:";
183     StringMap.iter print_vars value.fparameters;
184     print_endline "Local variables:";
185     StringMap.iter print_vars value.flocals;;
186
187 let print_interrupts key value =
188     print_string("Interrupt "^ key ^": \n");
189     print_endline "Local variables:";
190     StringMap.iter print_vars value.inlocals;;
191
192 let print_types key value =
193     print_string("Nifty50 Type "^ key ^": \n");
194     print_endline "Properties:";
195     StringMap.iter print_vars value.properties;
196     print_endline "Methods:";
197     StringMap.iter print_funcs value.functions;;
198
199 (*creates symbol tables for variables *)
200 let createNiftyVarSymbolTable map (var_elem: var_decl list) =
201   List.fold_left
202   (fun map thelist ->
203     if StringMap.mem thelist.vname map
204     then
205       raise(Failure("Compiler error: variable named \"" ^
    thelist.vname ^ "\" already exists."))
206     else
207       StringMap.add thelist.vname thelist.vtype map)
208   map var_elem
209
210 let createNiftyFuncSymbol (var_elem: func_decl) =
211   {
212     ftype = var_elem.rtype;
213     fparameters = createNiftyVarSymbolTable StringMap.empty
    var_elem.formals;
214     flocals = createNiftyVarSymbolTable StringMap.empty
    var_elem.locals;
215     fbodylist = var_elem.body;
216   }
217
218
219 (*creates symbol table for functions *)
```

```
220 let createNiftyFuncSymbolTable map (var_elem: func_decl list) =
221   List.fold_left (fun map thelist ->
222     if StringMap.mem thelist.fname map then
    raise(Failure("Compiler error: function named \'" ^ thelist.fname
    ^ "\' already exists.")) else
223     StringMap.add thelist.fname (createNiftyFuncSymbol thelist)
    map) map var_elem
224
225 (*creates symbol table for interrupts *)
226 let createNiftyInterruptSymbol (var_elem: interrupt_decl) =
227   {
228     inlocals = createNiftyVarSymbolTable StringMap.empty
    var_elem.ilocals;
229     inbodylist = var_elem.ibody;
230   }
231
232 (*creates symbol table for interrupts *)
233 let createNiftyInterruptSymbolTable map (var_elem: interrupt_decl
    list) =
234   List.fold_left (fun map thelist ->
235     if StringMap.mem thelist.iname map then
    raise(Failure("Compiler error:Interrupt handler named \'" ^
    thelist.iname ^ "\' already exists.")) else
236     StringMap.add thelist.iname (createNiftyInterruptSymbol
    thelist) map) map var_elem
237
238 (*creates symbol table for new types *)
239 let createNiftyTypesSymbol (var_elem: type_decl) =
240   {
241     thetype = var_elem.ytype;
242     properties = createNiftyVarSymbolTable StringMap.empty
    var_elem.yproperties;
243     functions = createNiftyFuncSymbolTable StringMap.empty
    var_elem.yfunctions;
244   }
245
246 (*creates symbol table for new types*)
247 let createNiftyTypesSymbolTable map (var_elem: type_decl list) =
248   List.fold_left (fun map thelist ->
249     if StringMap.mem thelist.yname map then
    raise(Failure("Compiler error: Type named \'" ^ thelist.yname ^
```

```
    "\' already exists.")) else
250     StringMap.add thelist.yname (createNiftyTypesSymbol thelist)
    map) map var_elem
251
252
253 let header =
254 "#include <stdio.h>
255 #include <string.h>
256 #include <unistd.h>
257 #include <stdlib.h>
258 #include \"Util.h\"
259 #include \"Interrupt.h\"
260 #include \"Array.h\"
261 "
262
263 (* Translates program in AST form into bytecode program.*)
264 (* Includes exception handling. *)
265     let translate (globals, functions, interrupts, types) =
266
267     let var_table = createNiftyVarSymbolTable StringMap.empty
    globals in
268
269     let fun_table = createNiftyFuncSymbolTable StringMap.empty
    functions in
270
271     let interrupt_table = createNiftyInterruptSymbolTable
    StringMap.empty interrupts in
272
273     let type_table = createNiftyTypesSymbolTable StringMap.empty
    types in
274     (* StringMap.iter print_types type_table; *)
275     let env =
276     {
277       niftyVarSymbolTable = var_table;
278       niftyFuncSymbolTable = fun_table;
279       niftyInterruptSymbolTable = interrupt_table;
280       niftyTypesSymbolTable = type_table;
281
282     } in
283
284     print_endline header;
```

```
285     output_globals var_table env;
286     output_types type_table env;
287     output_interrupts interrupt_table env;
288     output_functions fun_table Void env;
289 
290 (*End of File*)
```

```
#include "Array.h"
#include <math.h>
#include <stdlib.h>
#include <stdio.h>
#include <pthread.h>

#define map(type) for (i=0;i<length;i++){function(type array + i);}
#define C_PORT    (unsigned char   *)
#define C_SHORT   (short           *)
#define C_USHORT  (unsigned short  *)
#define C_INT     (int             *)
#define C_UINT    (unsigned int    *)
#define C_LONG    (long            *)
#define C_ULONG   (unsigned long   *)
#define C_FLOAT   (float           *)
#define C_DOUBLE  (double          *)

void Array_Map(void *array, void (*function)(void * arg), int length, TYPE type)
{
  int i = 0;
  switch (type)
  {
    case PORT:
      map(C_PORT)
      break;
    case SHORT:
      map(C_SHORT)
      break;
    case USHORT:
      map(C_USHORT)
      break;
    case INT:
      map(C_INT)
      break;
    case UINT:
      map(C_UINT)
      break;
    case LONG:
      map(C_LONG)
      break;
    case ULONG:
      map(C_ULONG)
      break;
    case FLOAT:
      map(C_FLOAT)
      break;
    case DOUBLE:
      map(C_DOUBLE)
      break;
    default:
      map()
      break;
  }

}
```

```c
#include "Interrupt.h"
#include <signal.h>
#include <stdio.h>
#include <unistd.h>

void Interrupt_SetSignal(int sig, Interrupt *interrupt)
{
    interrupt->Source = sig;
    switch (interrupt->Source)
    {
        case -1:
            //ignore all
            signal(AINT, SIG_IGN);
            break;
        case 0:
            //restore default to all
            signal(AINT, SIG_DFL);
            break;
        case AINT:
            signal(AINT, interrupt->func);

            break;
    }
}
void Interrupt_SetISR(void (*isr)(int), Interrupt *interrupt)
{
    interrupt->func = isr;
}
```

```c
#include "Util.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define Util_Length(x) sizeof(x)

void * Util_Swap(void *object, int length)
{
    int i;
    char temp;
    char *array = (char *)object;
    int arraymax = length - 1;
    for (i = 0; i < (length / 2); i++)
    {
        temp = array[i];
        array[i] = array[arraymax - i];
        array[arraymax - i] = temp;
    }

    return object;
}

void * Util_Move(void * object, void *location, int length)
{
    return memmove(location, object, (size_t)length);
}
```

```
1 (* Nifty50.ml by Aamir Sarwar Jahan, COMS W4115*)
2
3 (* use the Ast.ml and Compile.ml file *)
4 type action = Ast | Compiler
5
6 let _ =
7   let action = if Array.length Sys.argv > 1 then
8     List.assoc Sys.argv.(1) [ ("-a", Ast);
9                               ("-c", Compiler)]
10  else Compiler in
11  let lexbuf = Lexing.from_channel stdin in
12  let program = Parser.program Scanner.token lexbuf in
13  match action with
14    Ast -> let listing = Ast.string_of_program program
15           in print_string listing
16  | Compiler -> Execute.execute_prog (Compiler.translate program)
17
18 (* End of File *)
```

```
OBJS = ast.cmo parser.cmo scanner.cmo compile.cmo seal.cmo

TARFILES = Makefile testall.sh scanner.mll parser.mly \
    ast.ml compile.ml seal.ml \

nifty50 : $(OBJS)
    ocamlc -o nifty50 $(OBJS)

scanner.ml : scanner.mll
    ocamllex scanner.mll

parser.ml parser.mli : parser.mly
    ocamlyacc parser.mly

%.cmo : %.ml
    ocamlc -c $<

%.cmi : %.mli
    ocamlc -c $<

nifty50.tar.gz : $(TARFILES)
    cd .. && tar czf nifty50/nifty50.tar.gz $(TARFILES:%=nifty50/%)

.PHONY : clean
clean :
    rm -f nifty50 parser.ml parser.mli scanner.ml testall.log \
    *.cmo *.cmi *.o *.out *.s *.diff

# Generated by ocamldep *.ml *.mli
ast.cmo:
ast.cmx:
bytecode.cmo: ast.cmo
bytecode.cmx: ast.cmx
compiler.cmo: bytecode.cmo ast.cmo
compiler.cmx: bytecode.cmx ast.cmx
execute.cmo: bytecode.cmo ast.cmo
execute.cmx: bytecode.cmx ast.cmx
parser.cmo: ast.cmo parser.cmi
parser.cmx: ast.cmx parser.cmi
parser.cmi: ast.cmo

```

42
43