OGEL:

A LEGO Mindstorm Robot Controller Language

Matt Kalish - Michael Ching - Gerardo Flores - Charles O'Donnell

13 May 2003

Table of Contents

C Complete Code Listing

Chapter 1

An Introduction to OGEL

The OGEL language is intended for the development of programs to be executed on Lego Mindstorm robots. A single OGEL program, which uses a simple syntax very similar to that of Java, can easily and efficiently generate native code for one or more Lego Mindstorm microprocessors at the same time. The purpose of OGEL will therefore be to offer programmers a more powerful alternative to the current development tools and interfaces available for coding these Mindstorm robots.

1.1 Background

Lego Mindstorm is a product which allows users to turn Lego building blocks into responsive, autonomous robots, able to interact with their environment via predefined and/or computer assisted responses. The basic components in building Lego Mindstorm robots include the RCX 1.0 controller, miniature motors, sensors, and regular Lego blocks. The RCX 1.0 runs on firmware that is uploaded along with the user program through an infrared tower to the infrared interface on the controller.

1.2 Related Work

Several environments currently exist which facilitate the programming of Lego Mindstorm robots. Current efforts range from drag and drop interfaces targeted at novice programmers to the Not Quite C (NQC) language targeted at more experienced C programmers. NQC contains a feature-rich library for interfacing with the RCX controller. This provides a well-developed foundation to use as a means of communicating with the RCX.

1.3 Goals of OGEL

OGEL is designed to merge the best traits of all currently available technology, creating a language that is both easy and intuitive to use, while providing superior power and control. It is designed to help developers rapidly design, prototype and simulate their robotic entities. OGEL will ultimately compile down to NQC code.

1.3.1 Object Oriented

An object oriented language allows for a wide variety of enhancements over the current programming architecture of NQC. By mimicking traits of an object-oriented language, we sizably extend both the ease of programming and the power of drag-and-drop and NQC development solutions. The object-oriented nature of OGEL will serve as a layer of abstraction that simplifies the development cycle for programmers. They will be able to better focus on the task at hand and not be concerned with the physical constraints of the hardware nor the exact location and intent of the underlying code. By separating the ideas of control and physical blocks, one can easily reuse the components in multiple situations.

1.3.2 Hierarchical Architecture

OGEL creates a hierarchical development environment that facilitates the user's understanding and organization of the program. Robots have similar procedures: transportation movement, non-transportation movements (e.g. arm swings), and physical sensor detection. OGEL recognizes these similarities in programming development and supports a variety of keywords that organize these similar robot requirements into distinct classes. Thus, a programmer will be able to quickly analyze his/her program and verify if all necessary parts have been included in the robot just by looking to see if a movement class, arm-swinging class or sensor detection class have been created.

1.3.3 Scalable

OGEL will also take advantage of the inherent scalable nature of our hierarchical object-oriented approach. Employing the same building block based approach, large-scale programming projects can be easily designed and implemented using preexisting code and frameworks. By taking the details of the hardware and hiding it behind the abstraction of OGEL, large-scale projects can more easily be segmented as well.

1.3.4 Flexible

Although our language is designed for the RCX 1.0 controller, it would be ignorant of us to ignore possible (and probable) advances in the actual hardware units. Any system-specific traits will be user configurable. The language should also be general enough to be able to describe any goal a programmer wants a robot to achieve. This can be done by refraining from overly controller-specific references, similar to the outdated 'register' keyword in C.

1.3.5 Rapid Development

OGEL provides several capabilities that facilitate the rapid development of user code, such as modular and reusable code. The increased amount of development in the arena of Lego Mindstorms for research and recreational purposes, leads to large amounts of

code being created every day. In addition, the similarity in the movements and reactions of robots to stimuli beg the question, “Why must programmers re-create the wheel?” OGEL’s design recognizes that it is geared toward a programming-savvy community, and a major goal of the language is to allow a task to be described and programmed with relative ease. High level abstractions and simple calls to standard library routines aid in this quest.

1.3.6 Event Driven and Concurrent Processing

OGEL’s second goal is to be able to leverage the development of user-defined movements and combine them to create better event-based programming. One popular area in the robotics community is the creation of a soccer team of robots that will compete against a team of international superstars by 2050. To achieve this goal, event based programming must become more elaborate, robust and specific. OGEL does not attempt to completely bridge this gap in development efforts, but recognizes it as an issue and addresses it accordingly.

A physical system can be seen as nothing more than stimulus and response. Ex: A robot touches a wall, it moves the other direction. The RCX is also inherently multitasking and can have multiple control flows processing at the same time. Events and message passing internally between these tasks and externally with the outside world would ease development and present a clean model of Event-Response control to the programmer. In addition, this ability to program simultaneous threads of logic is an invaluable resource to programmers as they will inevitably desire their robots to perform several tasks at a time. Although more advanced systems have rudimentary notion of 'events' (RCX 2.0) RCX 1.0 does not. A generalized notion of events and messaging is built into OGEL.

1.3.7 Legacy Support

There exist countless NQC programs, which are already written and yet could benefit from the features introduced in OGEL. There also may be many opportunities where a specific algorithm has already been worked out in NQC and it would be unreasonable to require it to be rewritten. Therefore, OGEL will be able to understand blocks of NQC source code, and allow for NQC-style library calls to be handled in the same manner as well.

Chapter 2

Tutorial of OGEL

2.1 Preliminary Information

OGEL, while being very powerful and versatile, was designed with ease of implementation in mind. This tutorial will make it abundantly obvious how time-efficient and programmer-friendly it is to code in OGEL.

2.2 A First Example

Imagine a scenario where you would like to program two separate Lego Mindstorm robots to each perform a task. Each robot is made up of two RCX 1.0 processors – one which controls the upper body, and one which controls the lower body. The programmer can accomplish this via a single OGEL program, something that is currently impossible in other languages.

Examine the following code:

```
// RCX Declarations.
rcx robotUpper {SENSOR_TOUCH touch, out arm}
rcx robotLower {SENSOR_TOUCH bumper, out leftLeg, out rightLeg}

// Message Declarations.
message alert1;
message alert2;
```

This code is essentially known as the header section of an OGEL program. Its purpose is to declare the RCX units that are to be used and the messages that they are eligible to pass and receive while executing. In the RCX declarations, one must describe the inputs and outputs controlled by each RCX unit – ie, robotUpper has an arm that it controls, and it receives input from a sensor.

Now the programmer is free to implement whatever tasks will be necessary to achieve the program's ultimate goal. Tasks can be utilized by any or all of the RCX units declared in the program's header. Functions can also be used to accomplish key parts of a task. Examine the following code:

```
// A task, makes the robot move forward until a bump is encountered, then stops it.
task runUntilBump(in bumper)(out leftLeg, out rightLeg)
{
        // Block of native NQC code.
        <%
        setPower (leftLeg, OUT_FULL);
        setPower (rightLeg, OUT_FULL);
        %>

        // When directive, stops run when the sensor gets activated (a bump occurs).
        when (bumpSensor == 1) do stopMovement();
}

// A function, stops the current robot's motion.
void stopMovement()()(out leftLeg, out rightLeg)
{
        nqc.Off (leftLeg);
        nqc.Off (rightLeg);
}
```

The task "runUntilBump" makes the robot move forward until a bump is felt by the sensor, then the function stopMovement() is called and the motors are shut off. Important things to notice here are

1) All native NQC methods are available to the programmer and must simply be prefixed by "nqc." as nqc.Off is. NQC code that you simply want to include must be escaped by a <% … %>.
2) The when directive controls the motion of the robot – it keeps the motors running under full power until the sensor changes. When it does, control flow moves to the function.
3) A function's port mappings must be a subset of the calling task's port mappings. Other integer parameters may be passed to a function and they would be specified in the first set of parenthesis in the declaration.
4) OGEL programs can contain as many functions and tasks as desired, allowing of course for inevitable memory constraints.

Each OGEL program will contain exactly one task called main(), which serves to assign tasks and functions to individual RCX units as needed. These tasks are automatically activated at the start of execution. Examine the following code:

```
// The main task.
task main()
{
        // RCX instantiations.
        rcx robotUpper TimTorso;
        rcx robotLower TimLegs;
        rcx robotUpper JerryTorso;
```

```
        rcx robotLower JerryLegs;

        // Add task calls.
        TimLegs.addTask(runUntilBump);
        JerryLegs.addTask(runUntilBump);
}
```

Note that the four initially declared RCX units are now instantiated. It is next required that the RCX's are assigned the tasks that they will be responsible for running. Note that it is important that the task has the port mappings that are supported by the RCX – otherwise the application will not make sense semantically, although it looks syntactically correct.

So, the first complete OGEL program is as follows:

```
// RCX Declarations.
rcx robotOneUpper {SENSOR_TOUCH touch, out arm}
rcx robotOneLower {SENSOR_TOUCH bumper, out leftLeg, out rightLeg}

// Message Declarations.
message alert1;
message alert2;

// A task, makes the robot move forward until a bump is encountered, then stops it.
task runUntilBump (in bumper)(out leftLeg, out rightLeg)
{
        // Block of native NQC code.
        <%
        setPower (leftLeg, OUT_FULL);
        setPower (rightLeg, OUT_FULL);
        %>

        // When directive, stops run when the sensor gets activated (a bump occurs).
        when (bumpSensor == 1) do stopMovement();
}

// A function, stops the current robot's motion.
void stopMovement()()(out leftLeg, out rightLeg)
{
        nqc.Off (leftLeg);
        nqc.Off (rightLeg);
}
// The main task.
task main()
{
        // RCX instantiations
```

```
        rcx robotUpper TimTorso;
        rcx robotLower TimLegs;
        rcx robotUpper JerryTorso;
        rcx robotLower JerryLegs;

        // Add task calls.
        TimLegs.addTask(runUntilBump);
        JerryLegs.addTask(runUntilBump);
}
```

When executed properly, it will make two separate robots power their motors and begin walking until a bump is encountered, at which time they stop. This could easily be a subset of a larger, more useful OGEL program, such as a "showdown" program where two robots begin in the center of a walled room, and they walk until the wall is hit, then turn and fire some projectile at the opposing robot. As you can see from the sheer size and capability of the program above, OGEL provides a very powerful tool for programming Lego Mindstorm robots..

2.3 Compiling and Running OGEL Source Code

Source code files will be identified with a ".ogel" extension. A few requirements are necessary for the NQC compiler to function. First and foremost, you must be able to invoke antlr.Tool from the current compilation space and be able to run a bash script. The bash script is used as the driver for the whole process. After a program is written in OGEL, compilation is initiated with the command "ogelc filename.ogel", where filename is the name of the source file. Assuming that no compile-time errors occur, the compile will produce a number of output files, with the extention ".nqc". The number of files generated by the compiler will correspond to the number of RCX units instantiated in the main task.

To ultimately run the code in a real-life LEGO Mindstorm robot, however, each of these target files will need to be compiled using an existing NQC compiler. A good NQC website, containing links to NQC compilers, can be found at: http://www.baumfamily.org/nqc/index.html. Once the NQC files are compiled, the code must be uploaded onto the RCX units. Check LEGO Mindstorm documentation for details on this stage. Upon uploading the programs to each RCX involved in the application, everything it prepared to run the OGEL program on the robot(s).

Chapter 3

Language Reference Manual

3.1 Grammar Notation

The grammatical symbols are introduced in stages throughout this document, presenting only the symbols relevant to each subsection. Italicized symbols represent non-terminals, while those in quotes are terminal. Concatenated symbols imply concatenated tokens, while the '|' character implies a choice. Other regular expression notation used include the Kleene *, Kleene +, and ? optional character. Parentheses group symbols and apply precedence. Finally, the symbols *letter* and *digit* are defined to represent characters in the Roman alphabet plus underscores and Arabic numerals.

3.2 Lexical Rules

3.2.1 Comments

OGEL supports two types of comments. The first form begins with /* and ends with */. This form of comment is allowed to cover multiple lines, but cannot be nested. The second type of comment begins with // and ends at a new line. Comments are ignored by the compiler and allow the programmer to document the source code.

3.2.2 Whitespace

Whitespace is defined as the ASCII space, horizontal tab and form feed characters, as well as line terminators and comments. As long as tokens are distinguishable and separated by at least one whitespace, the addition or subtraction of whitespace has no effect on the meaning of the program.

3.2.3 Tokens

There are several types of tokens: identifiers, keywords, constants, separators, and embedded NQC blocks. Whitespace is ignored except as a token separator. Whitespace is sometimes required to separate adjacent tokens that might otherwise be combined into one token (i.e. identifiers, keywords and constants). Token formation is greedy: the input

is searched for the longest string of characters that would constitute a token. Tokens are comprised of ASCII characters only.

3.2.4 Identifiers

Identifiers are used for variable, task, RCX, and function names. The first character must be an upper or lower case letter or the underscore character. The remaining part of the identifier can be a combination of letter numbers and an underscore, but may not be separated by whitespace.

3.2.5 Keywords

OGEL reserves certain identifiers from use by a programmer. These reserved words are all keywords and may not be used as identifiers. A list is below:

bool	in	return
break	int	task
const	message	time
do	out	true
else	pass	void
false	rcx	when
if	receive	while

3.2.6 Line Terminators

Input characters are divided into lines by line terminators. Lines are terminated by the ASCII characters carriage return or linefeed. The carriage return linefeed combination is counted as one terminator, not two.

3.2.7 Embedded NQC blocks

Embedded NQC blocks allow the programmer to reuse working code and insert it accordingly into the new program. Embedded NQC blocks begin with the character sequence <% and end with %>. The compiler assumes that the embedded NQC code is fully functional and will not issue any errors or warnings. The embedded code is not allowed to contain the string '%>', otherwise it would effectively prematurely end the block of embedded code.

3.2.8 Separators

The following ASCII characters are separators:

{ } [ ] ; : . ,

3.3 Program Structure

An OGEL robot control *program* consists of one *source* file and any number of *library* files which all adhere to the following format: At the top of the file there is a header block, followed by any number of tasks or functions in any order. A source file will contain *one* main function located after the header, while library files cannot contain a main function. The source file can *include* other library files so that their header definition, tasks, and functions are available for use. Libraries serve the purpose to group high-level goals, robot type information, or any other broad categories.
Programs describe goals which encompass the use of multiple robots and RCX units, and are automatically compiled down to many Not Quite C (NQC) source files, one for each RCX unit involved. These NCQ source files are guaranteed to be operational (except for embedded NQC blocks).

file → *includeDecls rcxDecls messageDecls functionList taskDeclaratonList mainDeclaration*

3.3.1 Header

A header section is included in every OGEL source and library file. The header's primary purpose is to organize the naming and defining of multiple RCXs, and to define messages which are used to communicate between RCX units. This facilitates the ability to extend control to multiple RCXs and allows for different RCX input and output port configurations. Source files can use any number of includes to load library files, while library files are prohibited from using such a keyword.

3.3.1.1 Includes

A source file may include any number of library files. The files to be included with a given source file will be declaring “#include filename”, where filename is to have a .lib extension.

3.3.1.2 RCX Declaration

The RCX declaration is the next part of the header and it associates an RCX type with a name. A header file may have multiple RCX definitions, but each RCX must have a unique name throughout all source and library files. Following the RCX declaration is a code block encapsulated by the separators: '{' and '}' which is where the Output and Input Port mappings are made. An RCX must declare at least one port.

rcxDecls → (*rcxDeclaration*)*
rcxDeclaration → “rcx” *rcxIdentifier* “{“(*inputPortMapping* | *outputPortMapping*)+ “}”
rcxIdentifier → IDENTIFIER

3.3.1.3 Output and Input Port mappings

Output and Input Port mappings are used to associate a name with each unique port. The RCX supports up to a maximum of three output and three input ports, limiting the number of ports the user can make in the RCX declaration block. An output port mapping is created by listing an output port= '$A','$B' or '$C' with a unique name within the current RCX declaration block. Similarly, an input port mapping is created by listing an Input Port= '$1','$2' or '$3' with a unique name.

outputPortMapping	→	*outPortMap* "=" (*OUTA* | *OUTB* | *OUTC*) ";"
inputPortMapping	→	*inPortMap* "=" (*IN1* | *IN2* | *IN3*) ";"
outPortMap	→	IDENTIFIER
inPortMap	→	IDENTIFIER

3.3.1.4 Message Declaration

A message declaration gives a name for a message that will be accessible by all RCXs. The message declaration is created by using the 'message' keyword coupled with a unique message name. All RCXs defined within the header have access to the message.

messageDecls	→	(*messageDeclaration*)*
messageDeclaration	→	"message" *messageIdentifier* ";"
messageIdentifier	→	IDENTIFIER

3.3.2 Tasks

A task declaration defines the most high-level objectives to be performed by a single RCX unit. These objectives can be accomplished through OGEL's event-driven features or through more conventional signal checking and function calling. A task is defined by the keyword **task** followed by the task's identifier and a parenthesized list of portmaps. Each portmap within the list is specified to be either of type **in** or **out**. There can be anywhere from zero to three specified in or out portmaps. A task code block is encapsulated by the separators '{' and '}'. The block can begin by the declaration of all **when** directives, followed by the declaration of all local variables. There can also be any combination of standard code, function calls, embedded NQC code, or **pass**/**receive** calls.

taskDeclarationList	→	(*task*)*
task	→	*taskDeclaration* "{" *taskBody* "}"
taskDeclaration	→	"task" *taskIdentifier inOutList*
taskBody	→	(*taskBodyStatements*)*
taskBodyStatements	→	*whenStatement* | *standardStatement* | *NQCBLOCK*
taskIdentifier	→	*IDENTIFIER*

3.3.2.1 Standard Code and Function Calls

Source code placed within task blocks, after *when* directives, is sequentially executed, beginning as soon as a task is activated. Lines may include any type of *Statements*, such as variable assignment incorporated with expression evaluation, which are terminated by the ';' separator. *Statements* also include **if** and **while** block statements, which can be used to check an in port mapping against a certain value once. Functions can be called simply by the function identifier, followed by a parenthesized list of arguments, comma separated. The port mappings for the task are automatically matched with those of the function definition. The port mappings for the function must be a subset of the port mapping for the task which must invoke it.

statementList → (*standardStatement*)*
standardStatement → *whileBlock*
| "do" "{" (*standardStatement*)* "}" "while" "(" *whileCondition* ")"
| "break" ";"
| "if" "(" *ifCondition* ")" "{" (*standardStatement*)* "}" *elseBlock*
| "return" ";"
| *functionCall* ";"
| *variableDeclaration*
| *variableAssignment*
| *variableDeclaredAssignment*
| *messageStatement*
| *nqcFunctionCall*
elseBlock → ("else" "{" *statementList* "}")?
whileBlock → "while" "(" *whileCondition* ")" "{" *whileStatements* "}"
whileStatements → (*standardStatement*)*
whileCondition → *condition*
ifCondition → *condition*
condition → *conditionStatement*
| (*PARALLEL* | *NOT* | *AND*) *conditionStatement*)*
| ("true" | "false")
conditionStatement → *expression* (*LESSTHAN* | *GREATERTHAN* | *LEQUAL* | *GEQUAL* | *CEQUAL* | *NEQUAL*) *expression*
expression → *sumExpr*
sumExpr → *prodExpr* ((*PLUS* | *DASH*) *prodExpr*)*
prodExpr → *powExpr* ((*STAR* | *SLASH* | *MOD*) *powExpr*)*
powExpr → *atom* (*POW atom*)?
atom → (*NUMBER* | *IDENTIFIER*)
variableDeclaredAssignment → *variableIdentifier* (*EQUALS* | *PLUSEQ* | *MINUSEQ* | *TIMESEQ* | *DIVIDEEQ*) *rValue* ";"
variableAssignment → *variableType variableIdentifier* "=" *rValue* ";"

variableDeclaration → *variableType variableIdentifier* ";"
variableType → ("int" | "const" | "time" | "bool")
variableIdentifier → *IDENTIFIER*
rValue → (*NUMBER* | *IDENTIFIER* | "true" | "false")

3.3.2.2 When Directives

When directives serve as a monitor for sensor (*in*) data and will call a given function when the 'when' expression evaluates to true. (This operates differently than unary, inline if-tests of in port maps in standard code.) The format is **when** followed by a boolean expression equating a in port map to a constant or variable, followed by a **do** *FunctionIdentifier* where the function identifier is followed by only a parenthesized list of arguments. The port mappings are again matched from the task. When directives operate concurrent to the task, and will call the specified function as long as the expression evaluates to true (i.e. the function will be called multiple times if the expression remains true). In the event of multiple 'when' declarations evaluating to true on a given RCX unit, the order the functions are called in is undefined.

whenStatement → "when" *whenCondition whenStatementDo* ";"
whenCondition → "(" *inPortMap* (*LESSTHAN* | *GREATERTHAN* | *LEQUAL* | *GEQUAL* | *CEQUAL* | *NEQUAL*) (*NUMBER* | *variableIdentifier*) ")"
whenStatementDo → "do" *functionCall*

3.3.2.3 Messaging

OGEL offers a superior method of message passing compared to NQC because it creates a formal communication standard that facilitates increased notification of events between multiple RCXs. OGEL's messaging architecture relies upon the message names created in the header file and the **pass** and **receive** keywords.

messageStatement → (*passMessage* | *recieveMessage*)

Pass allows a RCX to express that a certain event has occurred. The construct is to use the pass keyword in addition to the name of the message that is to be sent.

passMessage → "pass" *messageIdentifier* ";"

Receive allows an RCX to search for a particular message from another RCX. The construct is to use the receive keyword with the name of the message, followed by a code block encapsulated by the separators: '{' and '}' that contains the RCXs response once the message has been received. The receive is blocking until the specific identifier has been received.

receiveMessage → "receive" *messageIdentifier* ";"

3.3.3 Functions

A function is a block of code called from within a task or other function, which performs some activity. A function name can be named any legal identifier. Its declaration takes both int type parameters and in/out type parameters, which can then be used within the body of the function. Functions do not have the capability to return any values, and thus are defined using the void keyword.

By their nature, a function's primary utility is performing a physical action, such as making a sound or powering a motor in a certain direction. In other words, functions generally perform the lower-level actions required by a task or larger function. It is important to note that functions can only be called by tasks or other functions that have a matching in/out type parameter list. This is because when making a function call, the only parameters required to be explicitly stated will be those of type int. The in/out type variables will be implicitly inherited as parameters from the calling function or task.

In OGEL, we elected to do away with NQC's concept of subroutines, instead choosing to use only functions in their place. We felt that this design kept it much simpler for the coder, as the differences between functions and subroutines in NQC are quite minute. Furthermore, there were so many restrictions placed on the use of subroutines in NQC that we found them to be too burdensome to include in OGEL. However, we reserve the right to use subroutines in the compilation from OGEL source to NQC.

functionList	→	(*function*)*
function	→	*functionDeclaration* "{" *statementList* "}"
functionDeclaration	→	"void" *functionIdentifier parameterList inOutList*
functionCall	→	*functionIdentifier functionParamsPasng*
functionParamsPasng	→	"(" ((*NUMBER* | *IDENTIFIER*) ("," (*NUMBER* | *IDENTIFIER*))*)? ")"
functionIdentifier	→	*IDENTIFIER*
parameterList	→	"(" (*arg* ("," *arg*)*)? ")"
arg	→	("int" | "const") *IDENTIFIER*

3.3.4 main()

Every OGEL program starts with the execution of the main task, and it is therefore mandatory. Within the main task, previously defined RCXs and functions (whether within the file or in a separate header file) may be utilized. Its purpose is to associate RCXs with tasks in our event driven model and accomplish all the necessary type checking between disparate files in a directory. Central to the restructuring of the 'main' task is the fact that this new structure is scalable in that several control tasks may be associated with individual RCXs within one file. These tasks may therefore be bound to different RCXs and make use of their modularity without having to recreate the code blocks in separate files. To instantiate an RCX, one would simply state the RCX id

followed by a valid identifier.

RCX id identifier;

The "addtask(taskList)" method would then be used to associate an RCX with a task.

RCX_identifier.addTask(task_identifier);

Unlike traditional Not Quite C syntax, the calling of functions and listing of statements within main is not allowed as there is no direct link between the running main task and a particular RCX. Control structures such as loops would, therefore, have to be included in the logic of other tasks, which are called from main and would reside on an RCX. Main functions act primarily as an "init" task in which a configuration for a robot, or set of robots, is initiated and the inherent multi-tasking nature of the Lego RCXs may be taken advantage of.

mainDeclaration	→	"task" "main" "(" ")" "{" *mainBody* "}"
mainBody	→	*rcxInstantiations rcxAddTaskList*
rcxInstantiations	→	(*rcxInstantiation*)+
rcxInstantiation	→	"rcx" *rcxClass rcxInstanceIdentifier* ";"
rcxClass	→	*IDENTIFIER*
rcxInstanceIdentifier	→	*IDENTIFIER*
rcxAddTaskList	→	(*rcxAddTaskCall*)+
rcxAddTaskCall	→	*rcxInstanceIdentifier* "." "addTask" "(" *taskIdentifierList* ")" ";" *taskIdentifierList* ")" ";"

Chapter 4

Project Plan

4.1 Team Responsibilities

Group members were assigned primary responsibilities with some aspects of the project. Although a large portion of decision-making and other work was done together as a group, the following describes each group member's primary tasks:

Matt Kalish	Test File Creation, Documentation Creation
Michael Ching	Lexer Creation, Parser, Final Completion and Correctness of Semantic Analysis, Intermediate Testing Documentation
Charles O'Donnell	Entire Back-End, Glue Stages, Code Generation, Documentation
Gerardo Flores	Lexer, Parsing, Semantic Analysis, Intermediate Testing, Final Documentation, Creation of Final Presentation

4.2 Project Timeline

The project was developed under a strict, predetermined timeline for reaching specific milestones. It factors in the natural tendency for groups to put work off until near the deadline, however, every effort was made throughout the process to get ahead of schedule. Because of this, the project log ideally will show dates much sooner then the flexible ones declared here. The schedule used is as follows:

Rough language specification, Whitepaper complete	February 18th, 2003
Development environment and version control software chosen	March 1st, 2003

High-level syntax determined	March 21st, 2003
Grammar complete	March 24th, 2003
Language Reference Manual complete	March 27th, 2003
Parser completed	April 15th, 2003
Static-semantic checker done	May 1st, 2003
Back-end complete	May 8th, 2003
Final testing phase complete, compiler code finalized	May 11th, 2003
Finalize documentation, project complete	May 13th, 2003

4.3 Software Development Environment

This project will be developed on both Linux and Solaris using Java 1.3.1. CVS will be used as the version control software, and ANTLR will aid in code generation and semantic checking in several parts of our compiler. Testing will be carried out both in UNIX and in practice using actual RCX 1.0 processors on Lego Mindstorm robots.

4.4 Project Log

The following describes the actual dates that we completed each project milestone:

Rough language specification, Whitepaper complete	February 15th, 2003
Development environment and version control software chosen	February 20th, 2003
High-level syntax determined	March 20th, 2003
Grammar complete	March 22nd, 2003
Language Reference Manual complete	March 26th, 2003
Parser completed	April 14th, 2003
Static-semantic checker done	May 10th , 2003

Back-end complete	May 10th, 2003
Final testing phase complete, compiler code finalized	May 11th, 2003
Finalize documentation, project complete	May 12th, 2003

Chapter 5

Architectural Design

5.1 Architecture

The OGEL compiler contains all of the features you would expect to find in a generic compiler. Essentially, these are the lexer, parser, AST walker / static-semantic checker, code generator, and symbol table. The flow of the OGEL compiler's components can be seen in Figure 5.1 below.

This compiler will take an OGEL source code file as input, and output a file, or series of files, in the NQC target language. Note that in the preprocessor, any number of library files (with the ".lib" extention) may be included into the main source file. After the preprocessor has completed its function, the lexer and parser will receive a single master file, representative of the entire OGEL program. The lexer and parser were coded with the help of ANTLR, a tool that generates Java code for many compiler-related applications. This stage of the compiler will end in the creation of a representative abstract syntax tree, which will then be a perfect interface for static-semantic checking. By referencing the symbol table, the AST walker will ensure that the source code semantically makes sense by checking that variable names actually exist, etc. Once it is found that the source file is syntactically and semantically correct by this process, the structure is ready to enter the compiler's back-end, beginning with the UNI_OGEL code generator. This is essentially an intermediate representation that can span across all LEGO Mindstorm architectures (namely the RCX 1.0, RCX 2.0, CyberMaster, and Scout). From here, the compiler branches off and generates the architecture-specific representation. Note that for this project, only the RCX 1.0 code generator was implemented by the OGEL group. However, the groundwork is layed out for the easy addition of any other architecture. The dashed lines indicate those features not implemented in this project. Ultimately, the code generator will return the target code (in NQC) to a series of files, one for each RCX unit instantiated in the main task. These files will then be all set for NQC compilation, and finally upload onto the individual RCX units.

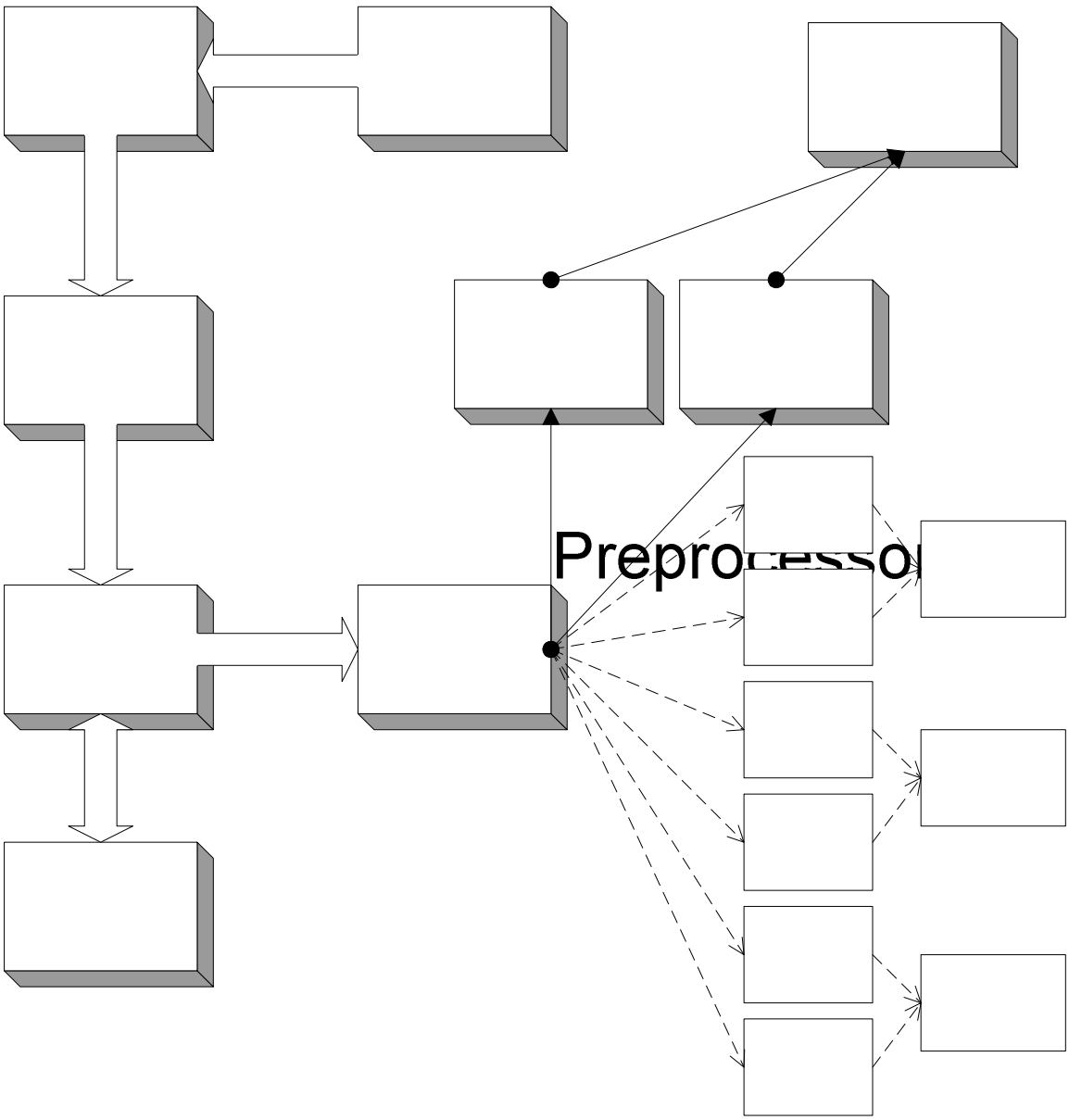

Figure 5.1 – The OGEL Compiler Flow Chart

5.2 Work Division

Many aspects of the compiler implementation were a combined group effort, at least as far as planning and overall organization are concerned. However, the major work on each module was done as follows:

Lexer / Parser

The lexer and parser were constructed mainly by Gerardo Flores and Michael Ching. The Static-Semantic Checker was a combined effort. The back-end / code generation was done mostly by Charles O'Donnell.

Chapter 6

Testing Plan

6.1 Goals

Our primary objective in test planning for the OGEL compiler is to create a well-thought out set of test cases from which we can deduce if errors exist in the code. While it is impossible to test all possible inputs for this implementation, it is certainly reasonable to develop a strategy to accurately locate and repair code bugs, which will ultimately lead to a final product that is sufficiently robust and reliable. Please note that this documents only the major testing sequences that the OGEL group performed – it will not include details on the frequent, extremely small-scale tests that naturally occur in the development of any computer program.

6.2 Hypothesis

Through use of thoughtfully created test cases, it will be trivial to prove the correctness of our implementation after any change, major or minor, in the code. If bugs do exist, we will not only be able to realize this, but also have insight as to where the specific errors are located in the code due to the highly specialized nature of the tests.

6.3 Methods

In general, the only way to effectively code a large project is to test each significant module as it is completed, and only when satisfied with the results should one continue on to the later stages. Keeping this in mind, the following test phases were utilized:

6.3.1 Phase I – Isolated Module Testing

In this phase, which will be performed after each module is completed, the group will test an individual subset of the OGEL compiler in isolation from other aspects of the project. These major subsets can be determined as the development process moves along, however a natural divide of modules far a generic compiler seems to be: lexer, parser, AST walker / static-semantic checker, and code generator. Certainly, as the OGEL group

deems necessary, these large subsets will be modularized into smaller portions that lend themselves better to isolation testing.

6.3.2 Phase II – Integration Testing

This is the second phase of the OGEL group's testing strategy, which will ensue after Phase I is complete for two connectable modules, and will then repeat after each iteration of Phase I until completion, where relevant and possible. This phase will be the first attempt to string together some sequence of modules into a unified program. Ideally, since all modules involved in the integration test should be completely functional, this phase will be brief. However, it is very necessary, not only to ensure that the program is progressing nicely, but also to catch any major errors made in the project planning. This must be a concern whenever a group gets together and divides the work up into "independent" pieces, and its better to catch such an error early rather than to complete all modules first to only then find out that they cannot be integrated. This phase will not handle large OGEL code examples, but rather complete skeletal source files that allow the group to ensure that the applicable sections of the compiler are in sync. When Phase II is completed successfully for an integration of all the project's modules, Phase III will commence.

6.3.3 Phase III – Final Testing

Phase III is the final stage, meant to test the compiler in real-life scenarios that can and will arise when developing OGEL programs. The OGEL group will ensure that every aspect of the OGEL compiler, from installation to execution, is satisfactory and as straightforward as possible. Phase III will also be the stage for elaborate OGEL programs to be tested. Ideally, the entire OGEL grammar will have been exercised by the time this phase is completed, and the group will be confident that the final compiler is the best one possible given the resources and time available for development.

6.4 Tools

The tests were performed manually for the most part. The use of makefiles and bash scripts automated the tast somewhat, but the actual test files were created such that the maximum amount of code coverage was achieved.

6.5 Implementation

The actual implementation of the OGEL group's testing plan proved to be challenging and tedious, but overall it made our development process run smoothly and relatively problem-free throughout the development cycle. Details of the testing follow.

6.5.1 Phase I – Isolated Module Testing

Phase I seeks to test each module independent of the other aspects of the compiler. This can be done by challenging each non-trivial method of the code thoroughly to ensure that they function as expected. Here is a brief overview:

1) Syntax Checker – To test the lexer and parser of our compiler (as generated by ANTLR), the group will run several test cases, spanning most of the OGEL grammar, through the module. The group will also run tests using several files with incorrect syntax. This phase of testing ceases only when all of the carefully selected test cases perform as expected when executed.

2) Static-Semantic Checker – For this stage, we will need to use a thorough sample of AST's, both correct and semantically erroneus, to test if the checker correctly identifies all semantic errors in the source file. For this stage, once we know the Syntax Checker is fully functional, we will probably move on to Phase II to do the majority of the testing by linking the modules together. This is because hand-constructing AST's and Symbol Tables is probably not a reasonable task. However, any and all subsets of the Static-Semantic Checker are eligible for isolation testing, and will be done as the OGEL group sees fit.

3) Code Generator – Here we need to be sure that the NQC program(s) outputted are representative of the OGEL source file. Again, moving to Phase II and III is probably the most effective way to test this on a large scale, but the group reserves the right to test any subset of this module in isolation.

6.5.2 Phase II – Integration Testing

In this stage, we elected to integrate the entire parser first (i.e., lexer and parser), ensuring that it was functional for all test cases presented. The group had previously considered these two components as one big module, but during implementation we found that they were quite distinct. Therefore, some time was spent integrating them together and therefore that integration had to be thoroughly tested. Although we were able to look at an AST representation, interpreting this result for each case was somewhat time-consuming and difficult. Therefore, the most meaningful results would not be found until the code generator was finally integrated.

Nevertheless, we spent time first integrating and testing the addition of the AST walker to the parser. The final stage of this testing phase was the addition of the code generator, which was last to be implemented. The group thoroughly tested the first completely integrated compiler with many skeletal examples, and after some debugging and additions to the code, the group felt confident that it was time for testing phase III.

6.5.3 Phase III – Final Testing

This final stage proved relatively easy, due to the fact that Phase II eliminated many of the major problems that existed in the OGEL compiler at that time. However, many problematic bugs and quirks in the code were ironed out by finally attempting to compile large sample programs.

After this stage was completed, the OGEL group felt confident that the compiler set forth was the best product possible given our constraints. The next section will contain two examples of successful compilations, and the target NQC code that resulted from this testing.

6.6 Test Case Compilation Results

The following two segments exhibit the work of our compiler on two significant test programs. First the OGEL source code will be shown, immediately followed by the target NQC files that result.

6.6.1 Test Case 1

6.6.2 Test Case 2

6.7 Work Division

Test plans with adequate code coverage developed by Matt Kalish. The large test cases were also written by Matt Kalish, with some smaller ones being written by each Gerardo Flores, Charles O'Donnell, and Micheal Ching as they were deemed necessary. The LEGO Mindstorm robots used in testing were constructed by Matt Kalish and Michael Ching. Actual Phase III testing was a cumulative group effort, with fixes being done by the author of the problematic module.

Chapter 7

Lessons Learned

7.1 Charles O'Donnell

7.2 Matt Kalish

The most important lesson I learned is how to effictively manage my time. I feel like the raw amount of time it takes to produce anything meaningful in Computer Science is so much that without having good skills in time management, it would be impossible to finish a project of this magnitude. Furthermore, I learned how easy it is for a group, even a small one, to have misunderstandings and communication problems. I respect the difficulty of small group work much more after participating on this project.

If I could give advice to future classes, I would say that the most important thing is to be honest about your abilities, and communicate problems you are having as soon as they occur. It only hurts your group if you claim to be able to do something you can't handle, or if you are too shy or embarrassed to admit when you are incapable of completing a certain task assigned to you.

7.3 Michael Ching

7.4 Gerardo Flores

As is usually the case, the most important thing in really making efficient use of the available time is keeping the lines of communication open. One must make sure that everyone is on the same page at all times. The most important lesson that I learned throughout this project is really how important "face time" is. It is not enough to simply instant message each other and assign tasks to one another, especially in a project of this magnitude. It is far too easy for people to shirk their tasks and never really contribute. Further, in this type of project, most of the stages are extremely interdependent and it is necessary to keep in constant contact so as not to damage the interface between stages.

Appendix A

Complete OGEL Grammar

file	→	*rcxDecls messageDecls functionList taskDeclaratonList mainDeclaration*
messageDecls	→	(*messageDeclaration*)*
messageDeclaration	→	"message" *messageIdentifier* ";"
messageStatement	→	(*passMessage* | *recieveMessage*)
passMessage	→	"pass" *messageIdentifier* ";"
receiveMessage	→	"receive" *messageIdentifier* ";"
messageIdentifier	→	*IDENTIFIER*
taskDeclarationList	→	(*task*)*
task	→	*taskDeclaration* "{" *taskBody* "}"
taskDeclaration	→	"task" *taskIdentifier inOutList*
taskBody	→	(*taskBodyStatements*)*
taskBodyStatements	→	*whenStatement* | *standardStatement* | *NQCBLOCK*
taskIdentifierList	→	(*taskIdentifier* ("," *taskIdentifier*)*)+
taskIdentifier	→	*IDENTIFIER*
whenStatement	→	"when" *whenCondition whenStatementDo* ";"
whenCondition	→	"(" *inPortMap* (*LESSTHAN* | *GREATERTHAN* | *LEQUAL* | *GEQUAL* | *CEQUAL* | *NEQUAL*) (*NUMBER* | *variableIdentifier*) ")"
whenStatementDo	→	"do" *functionCall*
mainDeclaration	→	"task" "main" "(" ")" "{" *mainBody* "}"
mainBody	→	*rcxInstantiations rcxAddTaskList*
rcxInstantiations	→	(*rcxInstantiation*)+
rcxInstantiation	→	"rcx" *rcxClass rcxInstanceIdentifier* ";"
rcxClass	→	*IDENTIFIER*
rcxInstanceIdentifier	→	*IDENTIFIER*
rcxAddTaskList	→	(*rcxAddTaskCall*)+
rcxAddTaskCall	→	*rcxInstanceIdentifier* "." "addTask" "(" *taskIdentifierList* ")" ";"
rcxDecls	→	(*rcxDeclaration*)*
rcxDeclaration	→	"rcx" *rcxIdentifier* "{"(*inputPortMapping* | *outputPortMapping*)+ "}"
rcxIdentifier	→	*IDENTIFIER*
outputPortMapping	→	*outPortMap* "=" (*OUTA* | *OUTB* | *OUTC*) ";"
inputPortMapping	→	*inPortMap* "=" (*IN1* | *IN2* | *IN3*) ";"
outPortMap	→	*IDENTIFIER*
inPortMap	→	*IDENTIFIER*

functionList → (*function*)*
function → *functionDeclaration* “{“ *statementList* “}”
functionDeclaration → “void” *functionIdentifier parameterList inOutList*
functionCall → *functionIdentifier functionParamsPasng*
functionParamsPasng → “(“ ((*NUMBER* | *IDENTIFIER*) (“,” (*NUMBER* | *IDENTIFIER*))*)? “)”
functionIdentifier → *IDENTIFIER*
nqcFunctionCall → *NQCFUNCTION IDENTIFIER LPAREN RPAREN*
statementList → (*standardStatement*)*
standardStatement → *whileBlock*
| "do" “{“ (*standardStatement*)* “}” "while" “(“ *whileCondition* “)”
| “break” “;”
| "if” “(“ *ifCondition* “)” “{“ (*standardStatement*)* “}” *elseBlock*
| "return" “;”
| *functionCall* “;”
| *variableDeclaration*
| *variableAssignment*
| *variableDeclaredAssignment*
| *messageStatement*
| *nqcFunctionCall*
elseBlock → ("else" “{“ *statementList* “}”)?
whileBlock → "while" “(“ *whileCondition* “)” “{“ *whileStatements* “}”
whileStatements → (*standardStatement*)*
whileCondition → *condition*
ifCondition → *condition*
condition → *conditionStatement*
| (*PARALLEL* | *NOT* | *AND*) *conditionStatement*)*
| ("true" | "false")
conditionStatement → *expression* (*LESSTHAN* | *GREATERTHAN* | *LEQUAL* | *GEQUAL* | *CEQUAL* | *NEQUAL*) *expression*
expression → *sumExpr*
sumExpr → *prodExpr* ((*PLUS* | *DASH*) *prodExpr*)*
prodExpr → *powExpr* ((*STAR* | *SLASH* | *MOD*) *powExpr*)*
powExpr → *atom* (*POW atom*)?
atom → (*NUMBER* | *IDENTIFIER*)
parameterList → “(“ (*arg* (“,” *arg*)*)? “)”
inOutList → *inList outList*
inList → “(“ “in” *inPortMap* (“,” “in” *inPortMap*)*)? “)”
outList → ((“out” *outPortMap* (“,” “out” *outPortMap*)*)? “)”
arg → (“int” | “const”) *IDENTIFIER*
variableDeclaredAssignment → *variableIdentifier* (*EQUALS* | *PLUSEQ* | *MINUSEQ* | *TIMESEQ* | *DIVIDEEQ*) *rValue* “;”
variableAssignment → *variableType variableIdentifier* “=” *rValue* “;”

variableDeclaration → *variableType variableIdentifier* “;”
variableType → ("int" | "const" | "time" | "bool")
variableIdentifier → *IDENTIFIER*
rValue → (*NUMBER* | *IDENTIFIER* | “true” | “false”)
PERIOD → “.”
POUND → “#”
PLUS → “+”
DASH → “-“
SLASH → “/”
STAR → “*”
PARALLEL → “||”
LESSTHAN → “<”
GREATERTHAN → “>”
COMMA → ”,”
EQUALS → ”=”
SEMI → “;”
COLON → “:”
COLEQUALS → “:=”
LPAREN → “(“
RPAREN → “)”
LBRACKET → “[“
RBRACKET → “]”
QUESTION → “?”
DQUESTION → “??”
LEQUAL → “<=”
GEQUAL → “>=”
NEQUAL → “!=”
CEQUAL → “==”
NOT → “!”
AND → “&&”
IMPLIES → “=>”
PLUSEQ → “+=”
MINUSEQ → ”-=”
TIMESEQ → “*=”
DIVIDEEQ → “/=”
NQCFUNCTION → “nqc.”
LCURLY → “{“
RCURLY → “}”
IN1 → “$1”
IN2 → “$2”
IN3 → “$3”
OUTA → “$A”
OUTB → “$B”
OUTC → “$C”
LIBRARYENDING → “.lib”
IDENTIFIER → (“a”..”z”|”A”..”Z”|”_”) (“a”..”z”|”A”..”Z”|”_”|”0”..”9”)*

NUMBER	→	(“0”..”9”)+
WS	→	(“ “ | “\t” | “\n” | “\r”)
SL_COMMENT	→	"//" (~”\n”)* “\n”
ML_COMMENT	→	"/*" anythingButEnd1 “*/”
NQCBLOCK	→	"<%" anythingButEnd2 "%>"

Appendix B

Code Style Conventions

B.1 Introduction

This section is intended to guide this group's programmers via stylistic guidelines that should be followed while coding the OGEL compiler. While the group respects the individual's right to have their own programming style, and therefore chose to not restrict the minute points of coding, we feel that some conventions are necessary to produce a fluid, professional-looking final product. What follows are the code style conventions adopted by the OGEL group for this project.

B.2 General Principles

The code should be as human readable as possible, without compromising creativity. If an elegant (and therefore confusing) solution to a problem is implemented, it must be thoroughly documented in the source file, and / or a README. Consistency with indentation, formatting, and variable declaring is to be prioritized.

B.3 Tabs

Tabs are to be used mainly for the purpose of consistent indentation of source code. While caution should be used since tabs may be displayed differently in different editors, they are nonetheless a valid whitespace character and should be used as such.

B.4 Documentation Comments

Documentation must be valued as a necessary component of any compiler's source code files. Because of this, we will make every effort to thoroughly comment at least every method and variable in the code. In addition, interesting situations will be documented with the programmer's approach to the problem, and how a solution was implemented.

Appendix C

Complete Code Listing

Parser and Lexer

```
class OGELParser extends Parser;
options {
      buildAST = true;
      exportVocab = OGEL;
      k = 4;
      defaultErrorHandler = false;
      }
tokens{
      FILE;
      RCXDECLS;
      PARAMETERLIST;
      INLIST;
      OUTLIST;
      RCXINSTANTIATIONS;
      TASKIDENTIFIERLIST;
      MESSAGEDECLS;
      RCXADDTASKLIST;
      RCXSTARTTASKLIST;
      RCXSTOPTASKLIST;
      TASKDECLARATIONLIST;
      FUNCTIONLIST;
      FUNCTIONPARAMSPASSING;
      INCLUDEDECLS;
      WHILECONDITION;
      IFCONDITION;
      RCXINSTANTIATION;
      STANDARDSTATEMENT;
      VARIABLEDECLARATION;
      VARIABLEASSIGNMENT;
      FUNCTIONDECLARATION;
      FUNCTION;
      STATEMENTLIST;
      FUNCTIONCALL;
      TASKDECLARATION;
      WHILESTATEMENTS;
      TASK;
      TASKBODY;
      TASKBODYSTATEMENTS;
      VARIABLEDECLAREDASSIGNMENT;

      PASSMESSAGE;

      RECEIVEMESSAGE;
```

```
      MESSAGESTATEMENT;
      RCXADDTASKCALL;
      RCXSTOPTASK;
      WHENCONDITION;
      NQCFUNCTIONCALL;
      WHENSTATEMENTDO;
      NQCBLOCKSTATEMENT;
      NQCPARAMETERS;
      NQCPARAMETERS1;
      NQCPARAMETERS2;
}

file
      : rcxDecls messageDecls functionList
      taskDeclarationList mainDeclaration EOF!{#file = #([FILE,
"file"], file);}
      ;

//MESSAGING

messageDecls
      : (messageDeclaration)*
      {#messageDecls = #([MESSAGEDECLS, "messageDecls"],
messageDecls);}
      ;

messageDeclaration
      : "message"^ messageIdentifier SEMI!
      ;

messageStatement
      : (passMessage | receiveMessage)
      //{#messageStatement = #([MESSAGESTATEMENT, "messageStatement"],
messageStatement);}
      ;

passMessage
      : "pass" messageIdentifier SEMI!
      {#passMessage = #([PASSMESSAGE, "passMessage"], passMessage);}
      //{#PASSMESSAGE = #([PASSMESSAGE, "passMessage"], passMessage);}
      ;

receiveMessage
      : "receive" messageIdentifier SEMI!
      {#receiveMessage = #([RECEIVEMESSAGE, "receiveMessage"],
receiveMessage);}
      ;

messageIdentifier
      : IDENTIFIER
      ;

//TASK STUFF

taskDeclarationList
      : (task)*
```

```
      {#taskDeclarationList = #([TASKDECLARATIONLIST,
"taskDeclarationList"], taskDeclarationList);}
      ;

task
      : taskDeclaration LCURLY! taskBody RCURLY!
      {#task = #([TASK, "task"], task);}
      ;


taskDeclaration
      : "task"! taskIdentifier inOutList
      {#taskDeclaration = #([TASKDECLARATION, "taskDeclaration"],
taskDeclaration);}
      ;

taskBody
      : (taskBodyStatements)*
      {#taskBody = #([TASKBODY, "taskBody"], taskBody);}
      ;

taskBodyStatements
      : whenStatement
      | standardStatement
      ;


taskIdentifierList
      : (taskIdentifier (COMMA! taskIdentifier)*)+
      {#taskIdentifierList = #([TASKIDENTIFIERLIST,
"taskIdentifierList"], taskIdentifierList);}
      ;

taskIdentifier
      : IDENTIFIER
      ;

whenStatement
      : "when"^ LPAREN! whenCondition RPAREN! whenStatementDo SEMI!
      ;

whenStatementDo
      : "do"! functionCall
      {#whenStatementDo = #([WHENSTATEMENTDO, "whenStatementDo"],
whenStatementDo);}
      ;

whenCondition
      : whenConditionStatement
      (options { greedy = true;}
      : (PARALLEL | NOT | AND) whenConditionStatement)*
      {#whenCondition = #([WHENCONDITION, "whenCondition"],
whenCondition);}
      ;

whenConditionStatement
```

```
      :   inPortMap ( LESSTHAN^ | GREATERTHAN^ | LEQUAL^ | GEQUAL^ |
CEQUAL^ | NEQUAL^) ( NUMBER | variableIdentifier)
      ;



//MAIN

mainDeclaration
      : "task"! "main"^ LPAREN! RPAREN! LCURLY! mainBody RCURLY!
      ;

mainBody: rcxInstantiations rcxAddTaskList
      ;

rcxInstantiations
      : (rcxInstantiation)+
      {#rcxInstantiations = #([RCXINSTANTIATIONS, "rcxInstantiations"],
rcxInstantiations);}
      ;

rcxInstantiation
      : "rcx"! rcxClass rcxInstanceIdentifier SEMI!
      {#rcxInstantiation = #([RCXINSTANTIATION, "rcxInstantiation"],
rcxInstantiation);}
      ;

rcxClass
      : IDENTIFIER
      ;

rcxInstanceIdentifier
      : IDENTIFIER
      ;

rcxAddTaskList
      : (rcxAddTaskCall)+
      {#rcxAddTaskList = #([RCXADDTASKLIST,"rcxAddTaskList"],
rcxAddTaskList);}
      ;

rcxAddTaskCall
      : rcxInstanceIdentifier PERIOD! "addTask"! LPAREN!
taskIdentifierList RPAREN! SEMI!
      {#rcxAddTaskCall = #([RCXADDTASKCALL,"rcxAddTaskCall"],
rcxAddTaskCall);}
      ;


rcxStartTaskList
      : (rcxStartTask)+
      {#rcxStartTaskList = #([RCXSTARTTASKLIST, "rcxStartTaskList"],
rcxStartTaskList);}
      ;

rcxStartTask
```

```
      : rcxInstanceIdentifier PERIOD! "start"! LPAREN!
taskIdentifierList RPAREN! SEMI!
      ;

rcxStopTaskList
      : (rcxStopTask)+
      {#rcxStopTaskList = #([RCXSTOPTASKLIST, "rcxStopTaskList"],
rcxStopTaskList);}
      ;

rcxStopTask
      : rcxInstanceIdentifier PERIOD! "stop"^ LPAREN!
taskIdentifierList RPAREN! SEMI!
      {#rcxStopTask = #([RCXSTOPTASK, "rcxStopTask"], rcxStopTask);}
      ;


rcxDecls
      : (rcxDeclaration)*
      {#rcxDecls = #([RCXDECLS, "rcxDecls"], rcxDecls);}
      ;

rcxDeclaration
      : "rcx"^ rcxIdentifier LCURLY! (inputPortMapping |
outputPortMapping)+ RCURLY!
      ;

rcxIdentifier
      : IDENTIFIER
      ;


outputPortMapping
      : outPortMap EQUALS^ (OUTA|OUTB|OUTC) SEMI!
      ;

inputPortMapping
      :  IDENTIFIER inPortMap EQUALS^ (IN1|IN2|IN3) SEMI!
      ;

outPortMap
      : IDENTIFIER
      ;

inPortMap
      : IDENTIFIER
      ;



//FUNCTION STUFF

functionList
      : (function)*
      {#functionList = #([FUNCTIONLIST,"functionList"], functionList);}
      ;
```

```
function
      : functionDeclaration LCURLY! statementList RCURLY!
      {#function = #([FUNCTION, "function"], function);}
      ;

functionDeclaration
      : "void"! functionIdentifier parameterList inOutList
      {#functionDeclaration = #([FUNCTIONDECLARATION,
"functionDeclaration"], functionDeclaration);}
      ;

functionCall
      : functionIdentifier functionParamsPassing
      {#functionCall = #([FUNCTIONCALL, "functionCall"],
functionCall);}
      ;

nqcFunctionCall
      : NQCFUNCTION! IDENTIFIER LPAREN! (nqcParameters | nqcParameters1
| nqcParameters2) RPAREN!
      {#nqcFunctionCall = #([NQCFUNCTIONCALL, "nqcFunctionCall"],
nqcFunctionCall);}
      ;


nqcParameters2
      : NUMBER
      {#nqcParameters2 = #([NQCPARAMETERS2, "nqcParameters2"],
nqcParameters2);}
      ;


nqcParameters1
      : (NUMBER COMMA! NUMBER)
      {#nqcParameters1 = #([NQCPARAMETERS1, "nqcParameters1"],
nqcParameters1);}
      ;


nqcParameters
      : (outPortMap (PLUS! outPortMap)*)?
            {#nqcParameters = #([NQCPARAMETERS, "nqcParameters"],
nqcParameters);}
      ;

functionParamsPassing
      : LPAREN! ( (NUMBER|IDENTIFIER) ( COMMA! (NUMBER|IDENTIFIER) )*
)? RPAREN!
      {#functionParamsPassing  =
#([FUNCTIONPARAMSPASSING,"functionParamsPassing"],
functionParamsPassing); }
      ;

functionIdentifier
      : IDENTIFIER
      ;
```

```
statementList
      : (standardStatement)*
      {#statementList = #([STATEMENTLIST, "statementList"],
statementList);}
      ;

standardStatement
      : whileBlock
      | "do"^ LCURLY! (standardStatement)* RCURLY! "while"! LPAREN!
whileCondition RPAREN!
      | "break"^  SEMI!
      | "if"^ LPAREN! ifCondition RPAREN! LCURLY! (standardStatement)*
RCURLY! elseBlock
      | "return"^ SEMI!
      | functionCall SEMI!
      | variableDeclaration
      | variableAssignment
      | variableDeclaredAssignment
      | messageStatement
      | nqcFunctionCall SEMI!
      | nqcBlockStatement
      ;

nqcBlockStatement
      : NQCBLOCK
      {#nqcBlockStatement = #([NQCBLOCKSTATEMENT, "nqcBlockStatement"],
nqcBlockStatement);}
      ;


elseBlock
      :("else"^ LCURLY! statementList RCURLY!)?
      ;

whileBlock
      : "while"^ LPAREN! whileCondition RPAREN! LCURLY! whileStatements
RCURLY!
      ;

whileStatements
      : (standardStatement)*
      {#whileStatements = #([WHILESTATEMENTS, "whileStatements"],
whileStatements);}
      ;

whileCondition
      : condition
      {#whileCondition = #([WHILECONDITION, "whileCondition"],
whileCondition);}
      ;

ifCondition
      : condition
      {#ifCondition = #([IFCONDITION, "ifCondition"], ifCondition);}
      ;

condition
```

```
      : conditionStatement
      ( options { greedy = true;}
      : (PARALLEL | NOT | AND) conditionStatement)*
      | ("true" | "false")
      ;


conditionStatement
      :  expression ( LESSTHAN^ | GREATERTHAN^ | LEQUAL^ | GEQUAL^ |
CEQUAL^ | NEQUAL^) expression
      ;


expression
      : sumExpr
      ;

sumExpr
      : prodExpr ((PLUS^|DASH^) prodExpr)*
      ;

prodExpr
      : powExpr ((STAR^ | SLASH^ | MOD^) powExpr)*
      ;

powExpr
      : atom (POW^ atom)?
      ;

atom
      : NUMBER
      | IDENTIFIER
      ;

parameterList
      : LPAREN! ( arg (COMMA! arg)* )? RPAREN!
      { #parameterList = #([PARAMETERLIST,"parameterList"],
parameterList); }
      ;

inOutList
      :  inList outList
      ;

inList
      : LPAREN! ("in"! inPortMap (COMMA! "in"! inPortMap)* )? RPAREN!
      {#inList = #([INLIST, "inList"], inList);}
      ;

outList
      : LPAREN! ("out"! outPortMap (COMMA! "out"! outPortMap)* )?
RPAREN!
      { #outList = #([OUTLIST,"outList"], outList);}
      ;

arg
      : ("int"|"const") IDENTIFIER
```

```
      ;


variableDeclaredAssignment
      : variableIdentifier (EQUALS^ | PLUSEQ^ | MINUSEQ^ | TIMESEQ^ |
DIVIDEEQ^)
      rValue SEMI!
      {#variableDeclaredAssignment = #([VARIABLEDECLAREDASSIGNMENT,
"variableDeclaredAssignment"], variableDeclaredAssignment);}
      ;

variableAssignment: variableType variableIdentifier EQUALS^ rValue
SEMI!
      {#variableAssignment = #([VARIABLEASSIGNMENT,
"variableAssignment"], variableAssignment);}
      ;

variableDeclaration :  variableType variableIdentifier SEMI!
      {#variableDeclaration = #([VARIABLEDECLARATION,
"variableDeclaration"], variableDeclaration);}
      ;

variableType : ("int" | "const" | "time" | "bool")
      ;

variableIdentifier: IDENTIFIER
      ;

rValue: (NUMBER | IDENTIFIER | "true" | "false")
      ;

class OGELLexer extends Lexer;

options {
  k = 4;                          // Lookahead to distinguish, e.g., :
and := */
charVocabulary = '\3'..'\377';

  exportVocab = OGEL;         // Export these token types for tree
walkers
  testLiterals = false;         // Disable checking every rule against
keywords
}

PERIOD :        '.' ;
POUND :         '#' ;
PLUS :          '+' ;
DASH :          '-' ;
SLASH :         '/' ;
STAR :          '*' ;
PARALLEL :      "||" ;
LESSTHAN :      '<' ;
GREATERTHAN :   '>' ;
COMMA :         ',' ;
EQUALS :        '=' ;
SEMI :      ';' ;
COLON :         ':' ;
```

```
COLEQUALS :     ":=" ;
LPAREN :        '(' ;
RPAREN :        ')' ;
LBRACKET :      '[' ;
RBRACKET :      ']' ;
QUESTION :      '?' ;
DQUESTION :     "??" ;
LEQUAL :        "<=" ;
GEQUAL :        ">=" ;
NEQUAL :        "!=" ;
CEQUAL :          "==" ;
NOT :       '!'  ;
AND:              "&&" ;
IMPLIES :       "=>" ;
PLUSEQ:           "+=";
MINUSEQ:    "-=";
TIMESEQ:    "*=";
DIVIDEEQ:   "/=";
NQCFUNCTION:      "nqc.";
LCURLY :        '{' ;
RCURLY :        '}' ;
IN1:  "$1";
IN2:  "$2";
IN3:  "$3";
OUTA: "$A";
OUTB: "$B";
OUTC: "$C";


IDENTIFIER
options { testLiterals = true;}
      :  ('a'..'z'|'A'..'Z'|'_') ('a'..'z'|'A'..'Z'|'_'|'0'..'9')*
      ;

NUMBER
      : ('0'..'9')+
      ;

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


SL_COMMENT :
      "//"
      (~'\n')* '\n'
      { $setType(Token.SKIP); newline(); }
      ;


// multiple-line comments
ML_COMMENT: "/*"(options{generateAmbigWarnings=false;}
      :
      { LA(2)!='/' }? '*'
```

```
      |     '\r' '\n'         {newline();}
      |     '\r'              {newline();}
      |     '\n'              {newline();}
      |     ~('*'|'\n'|'\r')
      )*
      "*/"
            {$setType(Token.SKIP);}
      ;


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

Semantic Walker

```
header{
import java.util.*;
}

class OGELSemanticWalker extends TreeParser;
options {
      importVocab = OGEL;
      }
{
      //vectors that hold persistant information that's needed
throughout
      //semantic walking
      Vector RCXs = new Vector();
      Vector Funcs = new Vector();
      Vector Messages = new Vector();
      Vector Tasks = new Vector();
      Vector Main = new Vector();
      Vector NQCFuncs = new Vector();

}

file: #(FILE setRCXs setMessages setFunction setTask
setMainDeclaration)
      ;

//rule that traverses all "rcx" nodes under RCXDECLS
setRCXs
{//name of RCX class
    String className;
    //port mappings
```

```
    String port1 = null;
    String port2= null;
    String port3= null;

    String portA= null;
    String portB= null;
    String portC= null;
}
      : #(RCXDECLS
        (#("rcx" className=getClass


      {//count the number of in and out ports declared
       int inports = 0;
       int outports = 0;
      }
      (#(d:EQUALS c:IDENTIFIER
      {
      AST a;
      AST b;

      if(d.getNumberOfChildren() > 2){
            {

            a=c.getNextSibling();
            b=a.getNextSibling();

            }
      }
      else{
            a = c;
            b = a.getNextSibling();

      }

      if(d.getNumberOfChildren() > 2){

      String sensType = c.getText();
      boolean checkSens = false;

      if(sensType.equals("SENSOR_TOUCH")){
            checkSens = true;
      }
      if(sensType.equals("SENSOR_LIGHT")){
            checkSens = true;

      }
      if(sensType.equals("SENSOR_ROTATION")){
            checkSens = true;

      }
      if(sensType.equals("SENSOR_CELSIUS")){
            checkSens = true;

      }
      if(sensType.equals("SENSOR_FAHRENHEIT")){
            checkSens = true;
```

```
	}
	if(sensType.equals("SENSOR_PULSE")){
		checkSens = true;

	}
	if(sensType.equals("SENSOR_EDGE")){
		checkSens = true;

	}

	if(!checkSens){
		System.err.println("Error: Undefined sensor type
declaration: "+sensType+".   Please indicate SENSOR_TOUCH,
SENSOR_LIGHT, SENSOR_ROTATION, SENSOR_CELSIUS, SENSOR_FAHRENHEIT,
SENSOR_PULSE or SENSOR_EDGE");
		System.exit(1);
	}
	}


	String temp = b.getText();

	if(temp.equals("$1")){
		port1 = a.getText();
		inports++;
	}
	if(temp.equals("$2")){
		port2 = a.getText();
		inports++;
	}
	if(temp.equals("$3")){
		port3 = a.getText();
		inports++;
	}
	if(temp.equals("$A")){
		portA = a.getText();
		outports++;
	}
	if(temp.equals("$B")){
		portB = a.getText();
		outports++;
	}
	if(temp.equals("$C")){
		portC = a.getText();
		outports++;
	}
	//else the port is not $abc123
	else if(!temp.equals("$A") && !temp.equals("$B") &&
!temp.equals("$C")
	   && !temp.equals("$1") && !temp.equals("$2") &&
!temp.equals("$3")){
		System.err.println("Port misdeclared: "+a.getText()+
		"cannot be assigned to "+b.getText());
		System.exit(1);
	}
	//ends the String temp
```

```
      }))*
      //checks to see if too many ports have been declared
      {
            if((inports > 3) || (outports > 3)){
                  System.err.println("Too many ports declared");
                  System.exit(1);
            }
      }

      )

      //when traversed one RCX, just create a new RCX object with the
      //required port mapping and names and insert it into the global
      //RCXs vector which will hold all RCX objects
      {RCX temp = new RCX(className, port1, port2, port3, portA, portB,
        portC);
       RCXs.add(temp);
      }

      )*)
      ;

//returns the class name of the rcx
getClass returns [String name]
      {name = null;}
      :IDENTIFIER {name = #IDENTIFIER.getText();}
      ;

getPorts [String p1, String p2, String p3, String pA, String pB, String
pC]
      {//count the number of in and out ports declared
       int inports = 0;
       int outports = 0;
      }
      :(#(EQUALS a:IDENTIFIER {AST b=a.getNextSibling();}
      {
      String temp = b.getText();
      if(temp.equals("$1")){
            p1 = a.getText();
            inports++;
      }
      if(temp.equals("$2")){
            p2 = a.getText();
            inports++;
      }
      if(temp.equals("$3")){
            p3 = a.getText();
            inports++;
      }
      if(temp.equals("$A")){
            pA = a.getText();
            outports++;
      }
      if(temp.equals("$B")){
            pB = a.getText();
            outports++;
      }
```

```
      if(temp.equals("$C")){
            pC = a.getText();
            outports++;
      }
      //else the port is not $abc123
      else if(!temp.equals("$A") && !temp.equals("$B") &&
!temp.equals("$C")
         && !temp.equals("$1") && !temp.equals("$2") &&
!temp.equals("$3")){
            System.err.println("Port misdeclared: "+a.getText()+
            "cannot be assigned to "+b.getText());
            System.exit(1);
      }
      }//ends the String temp
      ))*
      //checks to see if too many ports have been declared
      {
            if((inports > 3) || (outports > 3)){
                  System.err.println("Too many ports declared");
                  System.exit(1);
            }
      }

      ;
//parses the messages that are declared and adds them to the Messages
vector
setMessages: #(MESSAGEDECLS (#("message" a:IDENTIFIER
      {Messages.add(a.getText()); }))*)
      ;

setFunction: #(FUNCTIONLIST (#(FUNCTION
      //for each function make a function object
      {Function f = new Function();}
      parseFunctDeclaration[f] parseStatementList[f]))* )
      ;

parseFunctDeclaration [Function f]
      : #(FUNCTIONDECLARATION a:IDENTIFIER
      {f.name = a.getText();}
       setParameterList[f] setInList[f] setOutList[f]
      {
      f.loadPortsAndParams();
      Funcs.add(f);}
      )
      ;

setParameterList [Function f]
      :#(PARAMETERLIST (("int"|"const") a:IDENTIFIER
{f.addParam(a.getText());})* )

      ;

setInList [Function f]
      :#(a:INLIST
      {
      if((a.getNumberOfChildren() >= 0) && (a.getNumberOfChildren() <
4) )
```

```
       {
            AST b = a.getFirstChild();
            for(int i = 0; i < a.getNumberOfChildren(); i++)
            {
            f.inports[i] = b.getText();
            b = b.getNextSibling();
            }
       }
      else{
            System.err.println("Error: too many input port mappings
defined: "+f.name);
            System.exit(1);
      }


      }
      )
      ;

setOutList [Function f]
      :#(a:OUTLIST
      {if((a.getNumberOfChildren() > 0) && (a.getNumberOfChildren() <
4) )
       {
            AST b = a.getFirstChild();
            for(int i = 0; i < a.getNumberOfChildren(); i++)
            {
            f.outports[i] = b.getText();
            b = b.getNextSibling();
            }
       }
      else{
            System.err.println("Error: too many output port mappings
defined: "+f.name);
            System.exit(1);
      }

      }
      )
      ;


parseStatementList [Function f]
      :#(a:STATEMENTLIST
      {int children = a.getNumberOfChildren();

      if(children > 0){
       AST b = a.getFirstChild();
            for(int i =0; i < children; i++){
                  String temp = b.getText();
                  if(temp.equals("while"))
                        parseWhileBlock(b, f);
                  else if(temp.equals("do"))
                        parseDo(b, f);
                  else if(temp.equals("if"))
                        parseIf(b, f);
                  else if(temp.equals("variableAssignment"))
```

```
                        parseVariableAssignment(b, f);
                  else if(temp.equals("variableDeclaredAssignment"))
                        parseVariableDeclaredAssignment(b, f);
                  else if(temp.equals("variableDeclaration"))
                        parseVariableDeclaration(b, f);
                  else if(temp.equals("functionCall"))
                        parseFunctionCall(b, f);
                  else if(temp.equals("passMessage"))
                        parsePassMessage(b);
                  else if(temp.equals("receiveMessage"))
                        parseReceiveMessage(b);
                  else if(temp.equals("nqcFunctionCall"))
                        parseNQCFunctionCall(b, f);
                  b = b.getNextSibling();
            }
      }
      })
      ;

parseNQCFunctionCall [Function f]
      : #(a:NQCFUNCTIONCALL
      {
      AST b = a.getFirstChild();
      AST params = b.getNextSibling();
      int numParams = params.getNumberOfChildren();


      String func = b.getText();
      NQC tempNQC = f.nqcFunctionList;
      String [] tempFunctions = tempNQC.functions;
      int length = tempFunctions.length;
      boolean found = false;
      int index = 0;
      for(int i=0; i < length; i++){
            String checkFunc = tempFunctions[i];
            if(func.equals(checkFunc)){
                  found = true;
                  index = i;
                  break;
            }
      }
      if(found == false){
            System.err.println("Attempting to call a non NQC function
as a NQC function: "+func+" in function: "+f.name);
            System.exit(1);
      }

      //must now verify the function
      //On function
      if(((index >= 0) && (index < 8)) || (index == 11))
            parseNQCFunctionRegular(params, f, index);

      if(((index == 10) || (index == 13) || (index == 15)) &&
(numParams == 0)){
            parseNQCFunctionNoParam(params, f, index);
            }
```

```
      if(((index == 10) || (index == 13) || (index == 15)) &&
(numParams != 0)){
            System.err.println("Cannot pass a paramter with NQC
function: "+f.nqcFunctionList.functions[index]+"()");
            System.exit(1);
      }
      if(index == 8){
            parseNQCFunctionPlaySound(params, f, index);
      }

      if(index == 9){
            parseNQCFunctionParam1(params, f, index);
      }
      if(((index == 12) || (index == 14))){
            int child = params.getNumberOfChildren();
            if( child != 1){
                  System.err.println("Must pass exactly one parameter
to NQC function: "+f.nqcFunctionList.functions[index]+"() in function:
"+f.name);
                  System.exit(1);
            }
            AST param2 = params.getFirstChild();
            int type = param2.getType();
            if(type == NUMBER)
                  parseNQCFunctionParam2(params, f,  index);
            if(type != NUMBER){
                  if(index == 12){
                        System.err.println("Cannot pass NQC function
Wait() variable: "+param2.getText()+" in function: "+f.name+".  Must
pass an 'int'");
                        System.exit(1);
                  }
                  if(index == 14){
                        System.err.println("Cannot pass NQC function
SetSleepTime() variable: "+param2.getText()+" in function: "+f.name+".
Must pass an 'int'");
                        System.exit(1);
                  }
            }
      }

      })
      ;


parseNQCFunctionParam2 [Function f, int index]
      :#(a:NQCPARAMETERS2
      {

      int children = a.getNumberOfChildren();
      if(children != 1){
            System.err.println("Error: Can only pass one parameter to
NQC function: "+f.nqcFunctionList.functions[index]+"() in function:
"+f.name);
            System.exit(1);
      }
      int type = a.getFirstChild().getType();
```

```
      if(!(type == NUMBER)){
            System.err.println("Error: Can only pass an int to NQC
function: "+f.nqcFunctionList.functions[index]+"() in function:
"+f.name);
            System.exit(1);
      }
      });


//takes 2 number parameters
parseNQCFunctionParam1 [Function f, int index]
      :#(a:NQCPARAMETERS1
      {

      int children = a.getNumberOfChildren();

      if(children != 2){
            System.err.println("Error: Must pass two parameters to
PlayTone()");
            System.exit(1);
      }
      int temp = a.getFirstChild().getType();
      if(!(temp==NUMBER)){
            System.err.println("Error: Parameter:
"+a.getFirstChild().getText()+" in function: "+f.name+" must be of type
int");
            System.exit(1);
      }
      AST b = a.getFirstChild().getNextSibling();
      int temp2 = b.getType();
      if(!(temp2==NUMBER)){
            System.err.println("Error: Parameter: "+b.getText()+" in
function: "+f.name+" must be of type int");
            System.exit(1);
      }
      });



//looking for one parameter, of type keyword for sound
parseNQCFunctionPlaySound [Function f, int index]
      :#(a:NQCPARAMETERS
      {

      int children = a.getNumberOfChildren();
      if(children != 1){
            System.err.println("Error: Must list one parameter for NQC
function: "+ f.nqcFunctionList.functions[index]+"() in function:
"+f.name);
            System.exit(1);
      }
      String sound = a.getFirstChild().getText();

      boolean check = false;
      if((sound.equals("SOUND_CLICK")) ||
(sound.equals("SOUND_DOUBLE_BEEP")) || (sound.equals("SOUND_DOWN")) ||
```

```
(sound.equals("SOUND_UP")) || (sound.equals("SOUND_LOW_BEEP")) ||
(sound.equals("SOUND_FAST_UP"))){
            check = true;
      }
      if(!check){
            System.err.println("Error: Incorrect PlaySound() parameter:
"+sound+" in function: "+f.name+".  Use: SOUND_CLICK,
SOUND_DOUBLE_BEEP, SOUND_DOWN, SOUND_UP, SOUND_LOW_BEEP,
SOUND_FAST_UP");
            System.exit(1);
      }
      });

//nqc function with no parameters
parseNQCFunctionNoParam [Function f, int index]
      :#(a:NQCPARAMETERS
      {

      int children = a.getNumberOfChildren();
      if(children > 0){
            System.err.println("Error: NQC function:
"+f.nqcFunctionList.functions[index]+" in function: "+f.name+" does not
take any parameters");
            System.exit(1);
      }
      })
      ;


//reads in a bunch of output ports
parseNQCFunctionRegular [Function f, int index]
      :#(a:NQCPARAMETERS
      {


      int children = a.getNumberOfChildren();
      if((children ==1) && (index == 11)){
            boolean check = false;
            String temp = a.getFirstChild().getText();
            if( (temp.equals("TX_POWER_LO")) ||
(temp.equals("TX_POWER_HI"))){
                  check = true;
            }
      if(!check){

            System.err.println("Error: Parameter:
"+a.getFirstChild().getText()+" in NQC function: SetTxPower() in
function: "+f.name+" must be TX_POWER_LO or TX_POWER_HI");
            System.exit(1);
      }
      }
      else{

      if((children < 1) || (children > 3)){
            System.err.println("Error: incorrect number of parameters
for NQC function: "+f.nqcFunctionList.functions[index]+"() in function:
"+f.name+".  Must have at least one or up to three output ports");
```

```
            System.exit(1);
      }
      AST temp = a.getFirstChild();
      for(int i=0; i < children; i++){
            String outPortPassed = temp.getText();
            boolean verify = false;
            for(int j=0; j < 3; j++){
                  if(f.outports[j] != null){
                        String tempOutPort = f.outports[j];
                        if(outPortPassed.equals(tempOutPort)){
                              verify = true;
                        }
                  }
            }
      if(!verify){
            System.err.println("Error: attempting to use undeclared
output port: "+outPortPassed);
            System.exit(1);
      }
      temp = temp.getNextSibling();
      }
      }
      })
      ;




parseWhileCondition [Function f]
      :#(a:WHILECONDITION
      {
      int numConditions = a.getNumberOfChildren();

      AST condition = a.getFirstChild();


      for(int i = 0; i < numConditions; i++){
            boolean verify = false;
            if((condition.getNumberOfChildren()) != 0)
            {

            String variable = condition.getFirstChild().getText();
            ListIterator itr = f.symbolTable.listIterator(0);
            Hashtable current = (Hashtable)f.symbolTable.getFirst();

            if(current.containsKey(variable))
                  verify = true;
            else{while( !current.containsKey(variable) && itr.hasNext()
){
                  current = (Hashtable)itr.next();
                  if(current.containsKey(variable)){
                        verify = true;
                        break;
                        }
                  }
            }
```

```
            if (verify == false){
                  System.err.println("Error: Variable "+variable+" has
not been defined");
                  System.exit(1);
            }
            }
      condition = condition.getNextSibling();
      }

      });



parseWhileBlock [Function f]
      :#(a:"while" {
      Hashtable t = new Hashtable();
      f.symbolTable.addFirst(t);
      AST whileCondition = a.getFirstChild();
      AST whileStatements = whileCondition.getNextSibling();

      int numConditions = whileCondition.getNumberOfChildren();

      AST condition = whileCondition.getFirstChild();

      parseWhileCondition(whileCondition, f);

      //whileStatement Checking
      int children = whileStatements.getNumberOfChildren();
      AST b = whileStatements.getFirstChild();
      for (int j = 0; j < children; j++){
            String temp = b.getText();
            if(temp.equals("while"))
                        parseWhileBlock(b, f);
                  else if(temp.equals("do"))
                        parseDo(b, f);
                  else if(temp.equals("if"))
                        parseIf(b, f);
                  else if(temp.equals("variableAssignment"))
                        parseVariableAssignment(b, f);
                  else if(temp.equals("variableDeclaredAssignment"))
                        parseVariableDeclaredAssignment(b, f);
                  else if(temp.equals("variableDeclaration"))
                        parseVariableDeclaration(b, f);
                  else if(temp.equals("functionCall"))
                        parseFunctionCall(b, f);
                  else if(temp.equals("passMessage"))
                        parsePassMessage(b);
                  else if(temp.equals("receiveMessage"))
                        parseReceiveMessage(b);
                  else if(temp.equals("nqcFunctionCall"))
                        parseNQCFunctionCall(b, f);
                  b = b.getNextSibling();
            }

      f.symbolTable.removeFirst();
      });
```

```
parseDo [Function f]
      :#(a:"do"
      {
      //add a new context
      Hashtable t = new Hashtable();
      f.symbolTable.addFirst(t);

      int children = a.getNumberOfChildren();
       AST b = a.getFirstChild();
            for(int i =0; i < children; i++){
                  String temp = b.getText();
                  if(temp.equals("while"))
                        parseWhileBlock(b, f);
                  else if(temp.equals("do"))
                        parseDo(b, f);
                  else if(temp.equals("if"))
                        parseIf(b, f);
                  else if(temp.equals("variableAssignment"))
                        parseVariableAssignment(b, f);
                  else if(temp.equals("variableDeclaredAssignment"))
                        parseVariableDeclaredAssignment(b, f);
                  else if(temp.equals("variableDeclaration"))
                        parseVariableDeclaration(b, f);
                  else if(temp.equals("functionCall"))
                        parseFunctionCall(b, f);
                  else if(temp.equals("whileCondition"))
                        parseWhileCondition(b, f);
                  else if(temp.equals("passMessage"))
                        parsePassMessage(b);
                  else if(temp.equals("receiveMessage"))
                        parseReceiveMessage(b);
                  else if(temp.equals("nqcFunctionCall"))
                        parseNQCFunctionCall(b, f);
                  b = b.getNextSibling();
            }
      f.symbolTable.removeFirst();
      })
      ;

parseIf [Function f]
      : #(a:"if" b:IFCONDITION
      {
      parseIfCondition(b, f);

      //add a new context
      Hashtable t = new Hashtable();
      f.symbolTable.addFirst(t);

      int children = a.getNumberOfChildren();
      boolean done = false;

      //if there is actually something inside the if like standard
      //statements then we traverse them.  else do nothing
      if(children > 1){
            b = b.getNextSibling();
            for(int i =0; i < children-1; i++){
```

```
                  String temp = b.getText();
                  if(temp.equals("while"))
                        parseWhileBlock(b, f);
                  else if(temp.equals("do"))
                        parseDo(b, f);
                  else if(temp.equals("if"))
                        parseIf(b, f);
                  else if(temp.equals("variableAssignment"))
                        parseVariableAssignment(b, f);
                  else if(temp.equals("variableDeclaredAssignment"))
                        parseVariableDeclaredAssignment(b, f);
                  else if(temp.equals("variableDeclaration"))
                        parseVariableDeclaration(b, f);
                  else if(temp.equals("functionCall"))
                        parseFunctionCall(b, f);
                  else if(temp.equals("nqcFunctionCall"))
                        parseNQCFunctionCall(b, f);
                  else if(temp.equals("else")){
                        f.symbolTable.removeFirst();
                        done=true;
                        parseElse(b, f);
                  }
                  else if(temp.equals("passMessage"))
                        parsePassMessage(b);
                  else if(temp.equals("receiveMessage"))
                        parseReceiveMessage(b);

                  b = b.getNextSibling();
            }

            }
      if(!done){f.symbolTable.removeFirst();}

      }
      )
      ;

parseElse [Function f]
      : #(a:"else" b:STATEMENTLIST
      {parseStatementList(b, f);}
      )
      ;

parseIfCondition [Function f]
      :#(a:IFCONDITION
      {
      int numConditions = a.getNumberOfChildren();
      AST condition = a.getFirstChild();

      for(int i = 0; i < numConditions; i++){
            boolean verify = false;
            if((condition.getNumberOfChildren()) != 0)
            {

            String variable = condition.getFirstChild().getText();
            ListIterator itr = f.symbolTable.listIterator(0);
```

```
            Hashtable current = (Hashtable)f.symbolTable.getFirst();

            if(current.containsKey(variable))
                  verify = true;
            else{while( !current.containsKey(variable) && itr.hasNext()
){
                  current = (Hashtable)itr.next();
                  if(current.containsKey(variable)){
                        verify = true;
                        break;
                        }
                  }
            }
            if (verify == false){
                  System.err.println("Error1: Variable "+variable+" has
not been defined");
                  System.exit(1);
            }
            }
      condition = condition.getNextSibling();
      }

      });

parseFunctionCall [Function f]
      : #(FUNCTIONCALL a:IDENTIFIER b:FUNCTIONPARAMSPASSING
      {
      //check to see if the function exists
      boolean found = false;
       Object[] fa = Funcs.toArray();
       for(int i = 0; i < fa.length ; i++){
            if( ((Function)fa[i]).name.equals(a.getText()) )
                  found = true;
      }
      if(found){

      //check for matching number of parameters
      //now check for matching number of parameters
            int index = 0;
            int funcSize = Funcs.size();
            for(int i=0; i < funcSize; i++){
                  Function tempFunc = (Function)Funcs.elementAt(i);
                  String functionName = tempFunc.name;
                  if(a.equals(functionName)){
                        index = i;
                  }
            }

            int numParams = b.getNumberOfChildren();

            Function tempFunc2 = (Function)Funcs.elementAt(index);
            int checkParam = tempFunc2.counter;


            if(numParams != checkParam){
                System.err.println("Error: mismatched parameters "+a);
                  System.exit(1);
```

```
            }


      int children = b.getNumberOfChildren();
      if(children != 0){
      AST c = b.getFirstChild();

      //iterate through each child and check that the parameter has
already
      //been defined



      for(int i =0; i < children; i++){
            if(c.getType() == IDENTIFIER){
                  boolean verify = false;
                  String variable = c.getText();
                  ListIterator itr = f.symbolTable.listIterator(0);
                  Hashtable current =
(Hashtable)f.symbolTable.getFirst();

                  if(current.containsKey(variable))
                        verify = true;

                  else{
                        while( !current.containsKey(variable) &&
itr.hasNext() ){
                              current = (Hashtable)itr.next();
                              if(current.containsKey(variable)){
                                    verify = true;
                                    break;
                              }
                        }
                  }


                  if (verify == false){
                        System.err.println("Error: Variable
"+variable+" has not been defined and therefore cannot be passed to a
function.");
                        System.exit(1);
                  }
            }
      c = c.getNextSibling();
      }

      }
      }
      else{
      System.err.println("Error: Undelcared function: "+a.getText());
      System.exit(1);
      }

      })
      ;
```

```
parseVariableDeclaration [Function f]
      : #(a:VARIABLEDECLARATION
      {
      //first get type and name
      AST b = a.getFirstChild();
      String type = b.getText();
      String variable = b.getNextSibling().getText();

      //make a new Variable with the type and name and see if
      //it's already in the context symbol table.  If it is, throw an
      //error, if not insert into context
      Variable v = new Variable(type, variable);
      Hashtable t = (Hashtable)f.symbolTable.getFirst();

      //if variable is already declared
      if(t.containsKey(variable)){
            System.err.println("Error: Variable "+variable+" already
declared");
            System.exit(1);
      }
      else
            t.put(variable, v);
      }
      )
      ;

parseVariableAssignment [Function f]
      : #(VARIABLEASSIGNMENT a:EQUALS
      {
      //get the three children which are type, variable name, and
      //value
      AST b = a.getFirstChild();
      String type = b.getText();
      b = b.getNextSibling();
      String name = b.getText();
      b = b.getNextSibling();
      String value = b.getText();

      //get the current context hashtable
      Hashtable current = (Hashtable)f.symbolTable.getFirst();

      //check to see if the type matches the value
      if(type.equals("int")){
            //if int is being assigned to another variable, then check
            //the variables type
            if(b.getType() == IDENTIFIER){

            //get the current context hashtable

            ListIterator itr = f.symbolTable.listIterator(0);

            boolean verify = false;
            while( !current.containsKey(value) && itr.hasNext() ){
                  current = (Hashtable)itr.next();
                  if(current.containsKey(value)){
                        verify = true;
                        break;
```

```
                  }
            }
            if(current.containsKey(value)){
            verify = true;
            }
            if (verify == false){
                  System.err.println("Error: Variable "+value+" has not
been defined");
                  System.exit(1);
            }
      }

            else if(value.equals("true")||value.equals("false")){
                  System.err.println("Error: Cannot assign a boolean
value to an int");
                  System.exit(1);
            }
      }
      if(type.equals("const")){
            //if variable is of type const int, it can only be assigned
            //to a number
            if(b.getType() != NUMBER){
                  System.err.println("Error: Cannot assign anything but
a constant to a const int");
                  System.exit(1);
            }
      }
      if(type.equals("time")){
            if(b.getType() != NUMBER){
                  System.err.println("Error: Cannot assign a non number
to a time variable");
                  System.exit(1);
            }
            else if ((b.getType() == NUMBER) &&
(Integer.parseInt(b.getText()) < 0)){
                  System.err.println("Error: Cannot assign a negative
number to a time variable");
                  System.exit(1);
            }

      }
      if(type.equals("bool")){
            if(!value.equals("true") && !value.equals("false")){
                  System.err.println("Error: Cannot assign a non
boolean value to a bool");
                  System.exit(1);
            }
      }

      //check to see if that variable name has already been used in
this
      //context.  If not, then make a new variable
      if( ((Hashtable)f.symbolTable.getFirst()).containsKey(name) ){
            System.err.println("Error: Cannot reassign variable
"+name);
            System.exit(1);
      }
```

```
      else{
            ((Hashtable)f.symbolTable.getFirst()).put(name, new
Variable(type, name));
      }

      }
      )
      ;

parseVariableDeclaredAssignment [Function f]
      :#(a:VARIABLEDECLAREDASSIGNMENT
      {
      AST b = a.getFirstChild();
      String variable = b.getFirstChild().getText();
      AST c =  b.getFirstChild().getNextSibling();
      String value = c.getText();

      //get the current context hashtable
      Hashtable current = (Hashtable)f.symbolTable.getFirst();

      ListIterator itr = f.symbolTable.listIterator(0);

      boolean verify = false;
      while( !current.containsKey(variable) && itr.hasNext() ){
            current = (Hashtable)itr.next();
            if(current.containsKey(variable)){
                  verify = true;
                  break;
            }
      }
      if(current.containsKey(variable)){
            verify = true;
            }
            if (verify == false){
                  System.err.println("Error: Variable "+variable+" has
not been defined");
                  System.exit(1);
            }

      else{
      String hashtype = ((Variable)current.get(variable)).type;

      //check to see if the previously declared type matches the value
      if(hashtype.equals("int")){
            //if int is being assigned to another variable, then check
            //the variables type
            if(c.getType() == IDENTIFIER){
                  //get the current context hashtable
                  current = (Hashtable)f.symbolTable.getFirst();



                  ListIterator itr2 = f.symbolTable.listIterator(0);

                  boolean verify2 = false;
```

```
                  while( !current.containsKey(value) && itr2.hasNext()
){
                        current = (Hashtable)itr2.next();
                        if(current.containsKey(value)){
                        verify2 = true;
                        break;
                        }

                  }
            if(current.containsKey(value)){
                  verify = true;
            }

            if (verify2 == false){
                  System.err.println("Error: Variable "+value+" has not
been defined");
                  System.exit(1);
            }


                  else{
                        Variable temp = (Variable)current.get(value);
                        if(!temp.type.equals("int")){
                              System.err.println("Error: Cannot assign
a non int value to variable "+variable);
                              System.exit(1);
                        }
                  }

            }
            else if(value.equals("true")||value.equals("false")){
                  System.err.println("Error: Cannot assign a boolean
value to a previously declared int");
                  System.exit(1);
            }
      }
      if(hashtype.equals("const")){
            //if variable is of type const int, it can only be assigned
            //to a number
            if(c.getType() != NUMBER){
                  System.err.println("Error: Cannot assign anything but
a constant to a const int");
                  System.exit(1);
            }
      }
      if(hashtype.equals("time")){
            if(c.getType() != NUMBER){
                  System.err.println("Error: Cannot assign a non number
to a time variable");
                  System.exit(1);
            }
      }
      if(hashtype.equals("bool")){
            if(!value.equals("true") && !value.equals("false")){
                  System.err.println("Error: Cannot assign a non
boolean value to a bool");
                  System.exit(1);
```

```
            }
      }

      }//matches the else which states that it has been previously
declared

      }
      )
      ;

setMainDeclaration
      :#(a:"main" parseRCXInstantiations parseRCXAddTaskList)
      ;

parseRCXInstantiations
      :#(a:RCXINSTANTIATIONS
      {
      //number of instantiations
      int children = a.getNumberOfChildren();

      //name of the class and variable
      String variable = "";

      //boolean used to see if the rcx class was found
      boolean found = false;

      //get the array of declared RCXs
      Object[] rcxs = RCXs.toArray();

      //get the RCXInstantiation
      AST inst = a.getFirstChild();
      String className = inst.getFirstChild().getText();


      //iterate through number of instantiations
      for(int j =0; j < children ; j++){
            if(j !=0){
                  inst = inst.getNextSibling();
                  className = inst.getFirstChild().getText();
            }
            //iterate through declared rcxs
            for(int i = 0; i < rcxs.length ; i++){
                  if( ((RCX)rcxs[i]).className.equals(className) ){
                        found = true;
                        variable =
inst.getFirstChild().getNextSibling().getText();
                        Main.add(new MainRCX(variable, (RCX)rcxs[i]));
                  }
            }
            //if the class name of the RCX that you're trying to
instantiate is not
            //defined, then throw an error
            if(!found){
                  System.err.println("Error: Cannot instantiate an RCX
that has not been defined");
                  System.exit(1);
            }
```

```
      found = false;

      }
      }
      )
      ;

parseRCXAddTaskList
      :#(a:RCXADDTASKLIST
      {int children = a.getNumberOfChildren();

      Object[] mainRCX = Main.toArray();

      //get each child and check its rcx name and the task that's being
added
      AST b = a.getFirstChild();

      for(int i =0; i <children; i++)
      {
            boolean foundRCX = false;
            String rcxVar = b.getFirstChild().getText();

            String addedTask = null;
            //iterate through the rcx's that have been declared and see
if it matches
            String retreivedName = null;

            for(int j = 0; j < mainRCX.length; j++)
            {
                  retreivedName = ((MainRCX)mainRCX[j]).variable;

                  if(retreivedName.equals(rcxVar) && !foundRCX){
                        foundRCX = true;

      parseTaskIdentifierList(b.getFirstChild().getNextSibling(),(MainR
CX)mainRCX[j] );
                  }
            }
            if(!foundRCX){
            System.err.println("Error: Cannot add a task to an RCX:
"+retreivedName+" that has not been defined");
                  System.exit(1);
            }
      b = b.getNextSibling();
      }

      }
      )
      ;

parseTaskIdentifierList [MainRCX rcx]
      :#(a:TASKIDENTIFIERLIST
      {
      //get the number of tasks that are trying to be added
      int children = a.getNumberOfChildren();

      //get the first child to start traversing
```

```
      AST b = a.getFirstChild();
      String child = b.getText();

      //grab the already declared tasks
      Object[] tasks = Tasks.toArray();

      //iterate for each child
      for(int i = 0; i < children; i++){
      boolean foundTask = false;
            int index = 0;
            //iterate through the tasks looking for child
            for(int k =0; k < tasks.length; k++){
                  //if we find a task with the specified name
                  if(((Task)tasks[k]).name.equals(child)){
                        foundTask = true;
                        index = k;

                  //found the task, now check that the ports match
                  //task port mappings must represent a subset of rcx
port mappings

                  //check inport mappings

                  for(int j=0; j < 3; j++){
                        String tempFuncIn =
((Task)tasks[k]).inports[j];

                        if(tempFuncIn != null){
                              boolean match = false;
                              String tempTaskIn = rcx.rcx.port1;

                              if(tempFuncIn.equals(tempTaskIn)){
                                    match = true;

                              }
                              tempTaskIn = rcx.rcx.port2;
                              if(tempFuncIn.equals(tempTaskIn)){
                                    match = true;

                              }
                              tempTaskIn = rcx.rcx.port3;
                              if(tempFuncIn.equals(tempTaskIn)){
                                    match = true;

                              }

                        if (!match){
                              System.err.println("Mismatched inport
mappings between RCX: "+rcx.rcx.className+" and task:
"+((Task)tasks[index]).name);
                              //System.exit(1);
                        }
                        }
                  }

                  //check outport mappings
```

```
                  for(int m=0; m < 3; m++){
                        String tempFuncIn =
((Task)tasks[k]).outports[m];

                        if(tempFuncIn != null){
                              boolean match = false;
                              String tempTaskIn = rcx.rcx.portA;
                              if(tempFuncIn.equals(tempTaskIn)){
                                    match = true;

                              }
                              tempTaskIn = rcx.rcx.portB;
                              if(tempFuncIn.equals(tempTaskIn)){
                                    match = true;

                              }
                              tempTaskIn = rcx.rcx.portC;
                              if(tempFuncIn.equals(tempTaskIn)){
                                    match = true;

                              }

                        if (!match){
                              System.err.println("Mismatched outport
mappings between RCX: "+rcx.rcx.className+" and task:
"+((Task)tasks[index]).name);
                              //System.exit(1);
                        }
                        }
                  }

                  //add the task to the vector
                  rcx.addTask(((Task)tasks[index]).name);
                  }
            }
            if(!foundTask){
                  System.err.println("Error:  Cannot add an undeclared
task to an rcx");
                  System.exit(1);
            }

      b = b.getNextSibling();
      if(b !=null)
            child = b.getText();
      }
      })
      ;


setTask: #(TASKDECLARATIONLIST (#(TASK
         //for each task make a new task
           { Task t = new Task();}
         parseTaskDeclaration[t] parseTaskBody[t]))*)
         ;

parseTaskDeclaration [Task t]
    : #(TASKDECLARATION a:IDENTIFIER
```

```
    {t.name = a.getText();}
      setTaskInList[t] setTaskOutList[t]
    {t.loadPorts();
    Tasks.add(t);}
      )
      ;

setTaskInList [Task t]
     :#(a:INLIST
      {if((a.getNumberOfChildren() > 0) && (a.getNumberOfChildren() <
4) )
       {
            AST b = a.getFirstChild();
            for(int i = 0; i < a.getNumberOfChildren(); i++)
            {
            t.inports[i] = b.getText();
            b = b.getNextSibling();
            }
       }
      }
      )
      ;

setTaskOutList [Task t]
    :#(a:OUTLIST
      {if((a.getNumberOfChildren() > 0) && (a.getNumberOfChildren() <
4) )
       {
            AST b = a.getFirstChild();
            for(int i = 0; i < a.getNumberOfChildren(); i++)
            {
            t.outports[i] = b.getText();
            b = b.getNextSibling();
            }
       }
      }
      )
      ;


parseTaskBody [Task t]
    : #(a:TASKBODY
    {
    int children = a.getNumberOfChildren();
    if(children > 0){
      AST b = a.getFirstChild();
      for (int i=0; i < children; i++){

          String temp = b.getText();
          if(temp.equals("when"))
            parseWhenCondition(b, t);
           else if(temp.equals("if"))
            parseIfTask(b, t);
           else if(temp.equals("do"))
            parseDoTask(b, t);
             else if(temp.equals("while"))
            parseWhileBlockTask(b, t);
```

```
            else if(temp.equals("variableDeclaredAssignment"))
              parseVariableDeclaredAssignmentTask(b, t);
            else if(temp.equals("variableAssignment"))
              parseVariableAssignmentTask(b, t);
            else if(temp.equals("variableDeclaration"))
              parseVariableDeclarationTask(b, t);
            else if(temp.equals("passMessage"))
              parsePassMessage(b);
            else if(temp.equals("receiveMessage"))
              parseReceiveMessage(b);
            else if(temp.equals("functionCall"))
              parseFunctionCallTask(b, t);
            else if(temp.equals("nqcFunctionCall"))
              parseNQCFunctionCallTask(b, t);
            //else if(temp.equals("nqcBlockStatement"))

            b = b.getNextSibling();
        }
      }
      })
      ;



parseNQCFunctionCallTask [Task t]
      : #(a:NQCFUNCTIONCALL
      {
      AST b = a.getFirstChild();
      AST params = b.getNextSibling();
      int numParams = params.getNumberOfChildren();


      String func = b.getText();
      NQC tempNQC = t.nqcFunctionList;
      String [] tempFunctions = tempNQC.functions;
      int length = tempFunctions.length;
      boolean found = false;
      int index = 0;
      for(int i=0; i < length; i++){
            String checkFunc = tempFunctions[i];
            if(func.equals(checkFunc)){
                  found = true;
                  index = i;
                  break;
            }
      }
      if(found == false){
            System.err.println("Attempting to call a non NQC function
as a NQC function: "+func+" in function: "+t.name);
            System.exit(1);
      }

      //must now verify the function
      //On function
      if(((index >= 0) && (index < 8)) || (index == 11))
            parseNQCFunctionRegularTask(params, t, index);
```

```
      if(((index == 10) || (index == 13) || (index == 15)) &&
(numParams == 0)){
            parseNQCFunctionNoParamTask(params, t, index);
            }

      if(((index == 10) || (index == 13) || (index == 15)) &&
(numParams != 0)){
            System.err.println("Cannot pass a paramter with NQC
function: "+t.nqcFunctionList.functions[index]+"() in task: "+t.name);
            System.exit(1);
      }
      if(index == 8){
            parseNQCFunctionPlaySoundTask(params, t, index);
      }

      if(index == 9){
            parseNQCFunctionParam1Task(params, t, index);
      }
      if(((index == 12) || (index == 14))){
            int child = params.getNumberOfChildren();
            if(child != 1){
                  System.err.println("Must pass exactly one parameter
to NQC function: "+t.nqcFunctionList.functions[index]+"() in task:
"+t.name);
                  System.exit(1);
            }
            AST param2 = params.getFirstChild();
            int type = param2.getType();
            if(type == NUMBER)
                  parseNQCFunctionParam2Task(params, t, index);
            if(type != NUMBER){
                  if(index == 12){
                        System.err.println("Cannot pass NQC function
Wait() with variable: "+param2.getText()+" in task: "+t.name+".  Must
pass an 'int'");
                        System.exit(1);
                  }
                  if(index == 14){
                        System.err.println("Cannot pass NQC function
SetSleepTime() variable: "+param2.getText()+" in task: "+t.name+".
Must pass an 'int'");
                        System.exit(1);
                  }
            }
      }

      })
      ;




parseNQCFunctionParam2Task [Task t, int index]
      :#(a:NQCPARAMETERS2
      {

      int children = a.getNumberOfChildren();
```

```
      if(children != 1){
            System.err.println("Error: Can only pass one parameter to
NQC function: "+t.nqcFunctionList.functions[index]+"() in task:
"+t.name);
            System.exit(1);
      }
      int type = a.getFirstChild().getType();
      if(!(type == NUMBER)){
            System.err.println("Error: Can only pass an int to NQC
function: "+t.nqcFunctionList.functions[index]+"() in task: "+t.name);
            System.exit(1);
      }
      });


//takes 2 number parameters
parseNQCFunctionParam1Task [Task t, int index]
      :#(a:NQCPARAMETERS1
      {

      int children = a.getNumberOfChildren();

      if(children != 2){
            System.err.println("Error: Must pass two parameters to NQC
function PlayTone() in task: "+t.name);
            System.exit(1);
      }
      int temp = a.getFirstChild().getType();
      if(!(temp==NUMBER)){
            System.err.println("Error: Parameter:
"+a.getFirstChild().getText()+" in NQC function:
"+t.nqcFunctionList.functions[index]+" in task: "+t.name+" must be of
type int");
            System.exit(1);
      }
      AST b = a.getFirstChild().getNextSibling();
      int temp2 = b.getType();
      if(!(temp2==NUMBER)){
            System.err.println("Error: Parameter: "+b.getText()+" in
NQC function: "+t.nqcFunctionList.functions[index]+" in task:
"+t.name+" must be of type int");
            System.exit(1);
      }
      });



//looking for one parameter, of type keyword for sound
parseNQCFunctionPlaySoundTask [Task t, int index]
      :#(a:NQCPARAMETERS
      {

      int children = a.getNumberOfChildren();
      if(children != 1){
            System.err.println("Error: Must list one parameter for NQC
function: "+ t.nqcFunctionList.functions[index]+"() in task: "+t.name);
            System.exit(1);
```

```
      }
      String sound = a.getFirstChild().getText();

      boolean check = false;
      if((sound.equals("SOUND_CLICK")) ||
(sound.equals("SOUND_DOUBLE_BEEP")) || (sound.equals("SOUND_DOWN")) ||
(sound.equals("SOUND_UP")) || (sound.equals("SOUND_LOW_BEEP")) ||
(sound.equals("SOUND_FAST_UP"))){
            check = true;
      }
      if(!check){
            System.err.println("Error: Incorrect parameter: "+sound+"
in NQC function PlaySound in task: "+t.name+".  Use: SOUND_CLICK,
SOUND_DOUBLE_BEEP, SOUND_DOWN, SOUND_UP, SOUND_LOW_BEEP,
SOUND_FAST_UP");
            System.exit(1);
      }
      });

//nqc function with no parameters
parseNQCFunctionNoParamTask [Task t, int index]
      :#(a:NQCPARAMETERS
      {

      int children = a.getNumberOfChildren();
      if(children > 0){
            System.err.println("Error: NQC function:
"+t.nqcFunctionList.functions[index]+" in task: "+t.name+" does not
take any parameters");
            System.exit(1);
      }
      })
      ;


//reads in a bunch of output ports
parseNQCFunctionRegularTask [Task t, int index]
      :#(a:NQCPARAMETERS
      {


      int children = a.getNumberOfChildren();
      if((children ==1) && (index == 11)){
            boolean check = false;
            String temp = a.getFirstChild().getText();
            if( (temp.equals("TX_POWER_LO")) ||
(temp.equals("TX_POWER_HI"))){
                  check = true;
            }
      if(!check){
            System.err.println("Error: Parameter:
"+a.getFirstChild().getText()+" in task: "+t.name+" must be
TX_POWER_LOW or TX_POWER_HI");
            System.exit(1);
      }
      }
      else{
```

```
      if((children < 1) || (children > 3)){
            System.err.println("Error: incorrect number of parameters
for NQC function: "+t.nqcFunctionList.functions[index]+"() in task:
"+t.name+".  Must have at least one or up to three output ports");
            System.exit(1);
      }
      AST temp = a.getFirstChild();
      for(int i=0; i < children; i++){
            String outPortPassed = temp.getText();
            boolean verify = false;
            for(int j=0; j < 3; j++){
                  if(t.outports[j] != null){
                        String tempOutPort = t.outports[j];
                        if(outPortPassed.equals(tempOutPort)){
                              verify = true;
                        }
                  }
            }
      if(!verify){
            System.err.println("Error: attempting to use undeclared
output port: "+outPortPassed);
            System.exit(1);
      }
      temp = temp.getNextSibling();
      }
      }
      })
      ;



//////////////



parseWhileConditionTask [Task t]
      :#(a:WHILECONDITION
      {
      int numConditions = a.getNumberOfChildren();

      AST condition = a.getFirstChild();


      for(int i = 0; i < numConditions; i++){
            boolean verify = false;
            if((condition.getNumberOfChildren()) != 0)
            {

            String variable = condition.getFirstChild().getText();
            ListIterator itr = t.symbolTable.listIterator(0);
            Hashtable current = (Hashtable)t.symbolTable.getFirst();

            if(current.containsKey(variable))
```

```
                  verify = true;
            else{while( !current.containsKey(variable) && itr.hasNext()
){
                  current = (Hashtable)itr.next();
                  if(current.containsKey(variable)){
                        verify = true;
                        break;
                        }
                  }
            }
            if (verify == false){
                  System.err.println("Error: Variable "+variable+" has
not been defined");
            }
            }
      condition = condition.getNextSibling();
      }

      });


parseWhileBlockTask [Task t]

      :#(a:"while" {
      Hashtable h = new Hashtable();
      t.symbolTable.addFirst(h);
      AST whileCondition = a.getFirstChild();
      AST whileStatements = whileCondition.getNextSibling();

      int numConditions = whileCondition.getNumberOfChildren();

      AST condition = whileCondition.getFirstChild();

      parseWhileConditionTask(whileCondition, t);

      //whileStatement Checking
      int children = whileStatements.getNumberOfChildren();
      AST b = whileStatements.getFirstChild();
      for (int j = 0; j < children; j++){
            String temp = b.getText();
            if(temp.equals("while"))
                        parseWhileBlockTask(b, t);
                  else if(temp.equals("do"))
                        parseDoTask(b, t);
                  else if(temp.equals("if"))
                        parseIfTask(b, t);
                  else if(temp.equals("variableAssignment"))
                        parseVariableAssignmentTask(b, t);
                  else if(temp.equals("variableDeclaredAssignment"))
                        parseVariableDeclaredAssignmentTask(b, t);
                  else if(temp.equals("variableDeclaration"))
                        parseVariableDeclarationTask(b, t);
                  else if(temp.equals("functionCall"))
                        parseFunctionCallTask(b, t);
                  else if(temp.equals("nqcFunctionCall"))
                        parseNQCFunctionCallTask(b, t);
                  else if(temp.equals("passMessage"))
```

```
                        parsePassMessage(b);
                  else if(temp.equals("receiveMessage"))
                        parseReceiveMessage(b);
                  b = b.getNextSibling();
            }

      t.symbolTable.removeFirst();
      });

parseDoTask [Task t]
      :#(a:"do"
      {
      //add a new context
      Hashtable h = new Hashtable();
      t.symbolTable.addFirst(h);

      int children = a.getNumberOfChildren();
       AST b = a.getFirstChild();
            for(int i =0; i < children; i++){
                  String temp = b.getText();
                  if(temp.equals("while"))
                        parseWhileBlockTask(b, t);
                  else if(temp.equals("do"))
                        parseDoTask(b, t);
                  else if(temp.equals("if"))
                        parseIfTask(b, t);
                  else if(temp.equals("variableAssignment"))
                        parseVariableAssignmentTask(b, t);
                  else if(temp.equals("variableDeclaredAssignment"))
                        parseVariableDeclaredAssignmentTask(b, t);
                  else if(temp.equals("variableDeclaration"))
                        parseVariableDeclarationTask(b, t);
                  else if(temp.equals("functionCall"))
                        parseFunctionCallTask(b, t);
                  else if(temp.equals("nqcFunctionCall"))
                        parseNQCFunctionCallTask(b, t);
                  else if(temp.equals("whileCondition"))
                        parseWhileConditionTask(b, t);
                  else if(temp.equals("passMessage"))
                        parsePassMessage(b);
                  else if(temp.equals("receiveMessage"))
                        parseReceiveMessage(b);
                  b = b.getNextSibling();
            }
      t.symbolTable.removeFirst();
      })
      ;



parseIfTask [Task t]
      : #(a:"if" b:IFCONDITION
      {
      parseIfConditionTask(b, t);

      //add a new context
      Hashtable h = new Hashtable();
```

```
      t.symbolTable.addFirst(h);

      int children = a.getNumberOfChildren();
      boolean done = false;

      //if there is actually something inside the if like standard
      //statements then we traverse them.  else do nothing
      if(children > 1){
            b = b.getNextSibling();
            for(int i =0; i < children-1; i++){
                  String temp = b.getText();
                  if(temp.equals("while"))
                        parseWhileBlockTask(b, t);
                  else if(temp.equals("do"))
                        parseDoTask(b, t);
                  else if(temp.equals("if"))
                        parseIfTask(b, t);
                  else if(temp.equals("variableAssignment"))
                        parseVariableAssignmentTask(b, t);
                  else if(temp.equals("variableDeclaredAssignment"))
                        parseVariableDeclaredAssignmentTask(b, t);
                  else if(temp.equals("variableDeclaration"))
                        parseVariableDeclarationTask(b, t);
                  else if(temp.equals("functionCall"))
                        parseFunctionCallTask(b, t);
                  else if(temp.equals("nqcFunctionCall"))
                        parseNQCFunctionCallTask(b, t);
                  else if(temp.equals("else")){
                        t.symbolTable.removeFirst();
                        done=true;
                        parseElseTask(b, t);
                  }
                  else if(temp.equals("passMessage"))
                        parsePassMessage(b);
                  else if(temp.equals("receiveMessage"))
                        parseReceiveMessage(b);

                  b = b.getNextSibling();
            }

            }
      if(!done){t.symbolTable.removeFirst();}

      }
      )
      ;


parseStatementListTask [Task t]
      : #(a:STATEMENTLIST
    {
    int children = a.getNumberOfChildren();
    if(children > 0){
      AST b = a.getFirstChild();
      for (int i=0; i < children; i++){
          String temp = b.getText();
          if(temp.equals("when"))
```

```
            parseWhenCondition(b, t);
           else if(temp.equals("if"))
            parseIfTask(b, t);
           else if(temp.equals("do"))
            parseDoTask(b, t);
             else if(temp.equals("while"))
            parseWhileBlockTask(b, t);
          else if(temp.equals("variableDeclaredAssignment"))
            parseVariableDeclaredAssignmentTask(b, t);
          else if(temp.equals("variableAssignment"))
            parseVariableAssignmentTask(b, t);
          else if(temp.equals("variableDeclaration"))
            parseVariableDeclarationTask(b, t);
          else if(temp.equals("passMessage"))
            parsePassMessage(b);
          else if(temp.equals("receiveMessage"))
            parseReceiveMessage(b);
         else if(temp.equals("functionCall"))
            parseFunctionCallTask(b, t);
           else if(temp.equals("nqcFunctionCall"))
            parseNQCFunctionCallTask(b, t);


           b = b.getNextSibling();
      }
    }
    })
    ;

parseElseTask [Task t]
      : #(a:"else" b:STATEMENTLIST
      {parseStatementListTask(b, t);}
      )
      ;

parseIfConditionTask [Task t]
      :#(a:IFCONDITION
      {
      int numConditions = a.getNumberOfChildren();
      AST condition = a.getFirstChild();

      for(int i = 0; i < numConditions; i++){
            boolean verify = false;
            if((condition.getNumberOfChildren()) != 0)
            {

            String variable = condition.getFirstChild().getText();
            ListIterator itr = t.symbolTable.listIterator(0);
            Hashtable current = (Hashtable)t.symbolTable.getFirst();

            if(current.containsKey(variable))
                  verify = true;
            else{while( !current.containsKey(variable) && itr.hasNext()
){
                  current = (Hashtable)itr.next();
                  if(current.containsKey(variable)){
                        verify = true;
```

```
                        break;
                        }
                  }
            }
            if (verify == false){
                  System.err.println("Error: Variable "+variable+" has
not been defined");
            }
            }
      condition = condition.getNextSibling();
      }

      });

parsePassMessage
      : #(a:PASSMESSAGE
      {
      AST intermediate = a.getFirstChild();
      AST c = (AST)intermediate.getNextSibling();
      String messageName = c.getText();
      int messageVecSize = Messages.size();
      boolean check = false;
      for(int i=0; i < messageVecSize; i++){
            String tempString = (String)Messages.elementAt(i);
            if(tempString.equals(messageName)){
                  check = true;
                  break;
            }
      }
      if(!check){
            System.err.println("Message identifier: "+messageName+"
:undefined");
            System.exit(1);
      }
      });

parseReceiveMessage
      : #(a:RECEIVEMESSAGE
      {
      AST intermediate = a.getFirstChild();
      AST c = (AST)intermediate.getNextSibling();
      String messageName = c.getText();
      int messageVecSize = Messages.size();
      boolean check = false;
      for(int i=0; i < messageVecSize; i++){
            String tempString = (String)Messages.elementAt(i);
            if(tempString.equals(messageName)){
                  check = true;
                  break;
            }
      }
      if(!check){
            System.err.println("Message identifier: "+messageName+"
:undefined");
            System.exit(1);
      }
      });
```

```
parseVariableDeclarationTask [Task t]
      : #(a:VARIABLEDECLARATION
      {
      //first get type and name
      AST b = a.getFirstChild();
      String type = b.getText();
      AST c= b.getNextSibling();
      String variable = c.getText();

      //make a new Variable with the type and name and see if
      //it's already in the context symbol table.  If it is, throw an
      //error, if not insert into context
      Variable v = new Variable(type, variable);
      Hashtable h = (Hashtable)t.symbolTable.getFirst();

      //if variable is already declared
      if(h.containsKey(variable)){
            System.err.println("Error: Variable "+variable+" already
declared");
            System.exit(1);
      }
      else
            h.put(variable, v);
      }
      )
      ;


parseVariableAssignmentTask [Task t]
      : #(VARIABLEASSIGNMENT a:EQUALS
      {
      //get the three children which are type, variable name, and
      //value
      AST b = a.getFirstChild();
      String type = b.getText();
      b = b.getNextSibling();
      String name = b.getText();
      b = b.getNextSibling();
      String value = b.getText();

      //get the current context hashtable
      Hashtable current = (Hashtable)t.symbolTable.getFirst();

      //check to see if the type matches the value
      if(type.equals("int")){
            //if int is being assigned to another variable, then check
            //the variables type
            if(b.getType() == IDENTIFIER){

            //get the current context hashtable

            ListIterator itr = t.symbolTable.listIterator(0);

            boolean verify = false;
            while( !current.containsKey(value) && itr.hasNext() ){
```

```
                  current = (Hashtable)itr.next();
                  if(current.containsKey(value)){
                        verify = true;
                        break;
                  }
            }
            if(current.containsKey(value)){
            verify = true;
            }
            if (verify == false){
                  System.err.println("Error: Variable "+value+" has not
been defined");
                  System.exit(1);
            }
      }

            else if(value.equals("true")||value.equals("false")){
                  System.err.println("Error: Cannot assign a boolean
value to an int");
                  System.exit(1);
            }
      }
      if(type.equals("const")){
            //if variable is of type const int, it can only be assigned
            //to a number
            if(b.getType() != NUMBER){
                  System.err.println("Error: Cannot assign anything but
a constant to a const int");
                  System.exit(1);
            }
      }
      if(type.equals("time")){
            if(b.getType() != NUMBER){
                  System.err.println("Error: Cannot assign a non number
to a time variable");
                  System.exit(1);
            }
      }
      if(type.equals("bool")){
            if(!value.equals("true") && !value.equals("false")){
                  System.err.println("Error: Cannot assign a non
boolean value to a bool");
                  System.exit(1);
            }
      }

      //check to see if that variable name has already been used in
this
      //context.  If not, then make a new variable
      if( ((Hashtable)t.symbolTable.getFirst()).containsKey(name) ){
            System.err.println("Error: Cannot reassign variable
"+name);
            System.exit(1);
      }
      else{
            ((Hashtable)t.symbolTable.getFirst()).put(name, new
Variable(type, name));
```

```
	}

	}
	)
	;


parseVariableDeclaredAssignmentTask [Task t]
	:#(a:VARIABLEDECLAREDASSIGNMENT
	{
	AST b = a.getFirstChild();
	String variable = b.getFirstChild().getText();
	AST c =  b.getFirstChild().getNextSibling();
	String value = c.getText();

	//get the current context hashtable
	Hashtable current = (Hashtable)t.symbolTable.getFirst();

	ListIterator itr = t.symbolTable.listIterator(0);

	boolean verify = false;
	while( !current.containsKey(variable) && itr.hasNext() ){
		current = (Hashtable)itr.next();
		if(current.containsKey(variable)){
			verify = true;
			break;
		}
	}
	if(current.containsKey(variable)){
		verify = true;
		}
		if (verify == false){
			System.err.println("Error: Variable "+variable+" has
not been defined");
			System.exit(1);
		}

	else{
	String hashtype = ((Variable)current.get(variable)).type;

	//check to see if the previously declared type matches the value
	if(hashtype.equals("int")){
		//if int is being assigned to another variable, then check
		//the variables type
		if(c.getType() == IDENTIFIER){
			//get the current context hashtable
			current = (Hashtable)t.symbolTable.getFirst();




			ListIterator itr2 = t.symbolTable.listIterator(0);

			boolean verify2 = false;
			while( !current.containsKey(value) && itr2.hasNext()
){
				current = (Hashtable)itr2.next();
```

```
                        if(current.containsKey(value)){
                        verify2 = true;
                        break;
                        }

                  }
            if(current.containsKey(value)){
                  verify = true;
            }

            if (verify2 == false){
                  System.err.println("Error: Variable "+value+" has not
been defined");
                  System.exit(1);
            }


                  else{
                        Variable temp = (Variable)current.get(value);
                        if(!temp.type.equals("int")){
                              System.err.println("Error: Cannot assign
a non int value to variable "+variable);
                              System.exit(1);
                        }
                  }

            }
            else if(value.equals("true")||value.equals("false")){
                  System.err.println("Error: Cannot assign a boolean
value to a previously declared int");
                  System.exit(1);
            }
      }
      if(hashtype.equals("const")){
            //if variable is of type const int, it can only be assigned
            //to a number
            if(c.getType() != NUMBER){
                  System.err.println("Error: Cannot assign anything but
a constant to a const int");
                  System.exit(1);
            }
      }
      if(hashtype.equals("time")){
            if(c.getType() != NUMBER){
                  System.err.println("Error: Cannot assign a non number
to a time variable");
                  System.exit(1);
            }
      }
      if(hashtype.equals("bool")){
            if(!value.equals("true") && !value.equals("false")){
                  System.err.println("Error: Cannot assign a non
boolean value to a bool");
                  System.exit(1);
            }
      }
```

```
      }//matches the else which states that it has been previously
declared

      }
      )
      ;

parseFunctionCallTask [Task t]
      : #(FUNCTIONCALL a:IDENTIFIER b:FUNCTIONPARAMSPASSING
      {
      //check to see if the function exists
      boolean found = false;
       Object[] fa = Funcs.toArray();
       for(int i = 0; i < fa.length ; i++){
            if( ((Function)fa[i]).name.equals(a.getText()) )
                  found = true;
      }
      if(found){

      //check for matching number of parameters
      //now check for matching number of parameters
            int index = 0;
            int funcSize = Funcs.size();
            for(int i=0; i < funcSize; i++){
                  Function tempFunc = (Function)Funcs.elementAt(i);
                  String functionName = tempFunc.name;
                  if(a.equals(functionName)){
                        index = i;
                  }
            }

            int numParams = b.getNumberOfChildren();

            Function tempFunc2 = (Function)Funcs.elementAt(index);
            int checkParam = tempFunc2.counter;

            if(numParams != checkParam){
                System.err.println("Error: mismatched parameters
function: "+a+" task: "+t.name);
                  System.exit(1);
            }



      //check the parameters that are being passed
      int children = b.getNumberOfChildren();
      if(children != 0){
      AST c = b.getFirstChild();

      //iterate through each child and check that the parameter has
already
      //been defined



      for(int i=0; i < children; i++){
            if(c.getType() == IDENTIFIER){
```

```
                  boolean verify = false;
                  String variable = c.getText();

                  //finding parameter names
                  ListIterator itr = t.symbolTable.listIterator(0);
                  Hashtable current =
(Hashtable)t.symbolTable.getFirst();

                  if(current.containsKey(variable))
                        verify = true;

                  else{
                        while( !current.containsKey(variable) &&
itr.hasNext() ){
                              current = (Hashtable)itr.next();
                              if(current.containsKey(variable)){
                                    verify = true;
                                    break;
                              }
                        }
                  }


                  if (verify == false){
                        System.err.println("Error: Variable
"+variable+" has not been defined and therefore cannot be passed to a
function.");
                        System.exit(1);
                  }
            }
      c = c.getNextSibling();
      }

      //now check whether the function's input port and output port
mappings are a subset of the tasks'

      for(int j=0; j < 3; j++){
                String tempFuncIn = tempFunc2.inports[j];
                if(tempFuncIn != null){
                  boolean match = false;
                  for(int m=0; m < 3; m++){
                      String tempTaskIn = t.inports[m];
                      if(tempFuncIn.equals(tempTaskIn)){
                        match = true;
                        break;
                      }
                  }
                  if (!match){
                      System.err.println("Mismatched inport mappings
between function: "+tempFunc2.name+" and task: "+t.name);
                        System.exit(1);
                  }
               }
            }

            //now check the outport mappings
            for(int n=0; n < 3; n++){
```

```
                String tempFuncIn = tempFunc2.inports[n];
                if(tempFuncIn != null){
                  boolean match2 = false;
                  for(int q=0; q < 3; q++){
                      String tempTaskIn = t.inports[q];
                      if(tempFuncIn.equals(tempTaskIn)){
                        match2 = true;
                        break;
                      }
                  }
                  if (!match2){
                      System.err.println("Mismatched outport mappings
between function: "+tempFunc2.name+" and task: "+t.name);
                        System.exit(1);
                        }
                }
            }

      }

      }
      else{
      System.err.println("Error: Undelcared function: "+a.getText());
      System.exit(1);
      }



      })
      ;

parseWhenCondition [Task t]
    : #(b:"when" a:WHENCONDITION
    {

      int numConditions = a.getNumberOfChildren();
      //grab the first condition
      AST condition = a.getFirstChild();
      //grab the do part of the when statement
      AST whenStatementDo = a.getNextSibling();

      //iterate through the when conditions checking if the inport
mapping
      //is an identifier and if it has been initialized

      for(int i = 0; i < numConditions; i++){
            boolean verify = false;
            if((condition.getNumberOfChildren()) != 0)
            {

            //find string representation of input port mapping
            String variable = condition.getFirstChild().getText();

            //iterate through all previous context's looking to see
            //if the variable has been used
            ListIterator itr = t.symbolTable.listIterator(0);
            Hashtable current = (Hashtable)t.symbolTable.getFirst();
```

```
            if(current.containsKey(variable)){
                  Variable tempVar = (Variable)current.get(variable);

                  String type = tempVar.type;
                  //throw an error if the identifier was associated
with an output
                  //port mapping
                  if(type.equals("out")){
                        System.err.println("Error: Cannot assign output
port mapping to when condition: "+variable);
                        System.exit(1);
                  }
            }
            if(current.containsKey(variable))
                  verify = true;
            else{ while( (!current.containsKey(variable)) &&
itr.hasNext()){
                  current = (Hashtable)itr.next();
                  if(current.containsKey(variable)){
                        verify = true;
                        break;
                        }
                  }

            }

            if (verify == false){
                  System.err.println("Error:  1 inport map "+variable+"
has not been defined");
                  System.exit(1);
            }


            }
      condition = condition.getNextSibling();
      }


      int doChildren = whenStatementDo.getNumberOfChildren();
      AST function = whenStatementDo.getFirstChild();

      //must check names of all function calls

      for(int i=0; i < doChildren; i++){
          boolean verify = false;
          if((whenStatementDo.getNumberOfChildren()) != 0){
            //variable is the name of the function being called
            String variable = function.getFirstChild().getText();
            int funcsLength = Funcs.size();
            //index represents where the function is stored in the
Funcs vector
            int index = -10;
            //iterate through the Funcs vector looking for a matching
name
            for (int k=0; k < funcsLength; k++){
                Function tempFunc = (Function)Funcs.elementAt(k);
                String tempName = tempFunc.name;
```

```
                if(variable.equals(tempName)){
                  verify = true;
                  index = k;
                  break;


                }
            }
            if (verify == false){
                System.err.println("Error: Attempting to call
undeclared function "+variable);
      System.exit(1);
            }


            //now check for matching number of parameters

            AST intermediate = function.getFirstChild();
            AST funcParamsPass = intermediate.getNextSibling();
            int numParams = funcParamsPass.getNumberOfChildren();

            Function tempFunc = (Function)Funcs.elementAt(index);

            if(numParams != 0){
                  System.err.println("Should not pass a function with
parameters to a when condition. Function: "+variable);
                  System.exit(1);
            }




      AST c = funcParamsPass.getFirstChild();

      //iterate through each child and check that the parameter has
already
      //been defined

      for(int p=0; p < numParams; p++){
            if(c.getType() == IDENTIFIER){
                  boolean verify3 = false;
                  String variable2 = c.getText();

                  //finding parameter names
                  ListIterator itr = t.symbolTable.listIterator(0);
                  Hashtable current =
(Hashtable)t.symbolTable.getFirst();

                  if(current.containsKey(variable2)){
                        verify3 = true;
                  }
                  else{
                        while( !current.containsKey(variable2) &&
itr.hasNext() ){
                              current = (Hashtable)itr.next();
                              if(current.containsKey(variable2)){
                                    verify3 = true;
```

```
                                    break;
                                }
                            }
                        }


                        if (verify3 == false){
                            System.err.println("Error: Variable
"+variable+" has not been defined and therefore cannot be passed to a
function.");
                            System.exit(1);
                        }
                    }
                c = c.getNextSibling();
                }



                //now check the inport and outport mappings
                //function port mappings must represent a subset of task
port mappings
                //check inport mappings


                for(int j=0; j < 3; j++){
                    String tempFuncIn = tempFunc.inports[j];
                    if(tempFuncIn != null){
                      boolean match = false;
                      for(int m=0; m < 3; m++){
                          String tempTaskIn = t.inports[m];
                          if(tempFuncIn.equals(tempTaskIn)){
                            match = true;
                            break;
                          }
                      }
                      if (!match){
                          System.err.println("Mismatched inport mappings
between function: "+variable+" and task: "+t.name);
                            System.exit(1);
                      }
                    }
                }

                //now check the outport mappings
                for(int n=0; n < 3; n++){
                    String tempFuncIn = tempFunc.inports[n];
                    if(tempFuncIn != null){
                      boolean match2 = false;
                      for(int q=0; q < 3; q++){
                          String tempTaskIn = t.inports[q];
                          if(tempFuncIn.equals(tempTaskIn)){
                            match2 = true;
                            break;
                          }
                      }
                      if (!match2){
```

```
                        System.err.println("Mismatched outport mappings
between function: "+variable+" and task: "+t.name);
                              System.exit(1);
                        }
                        }
                }
                function = function.getNextSibling();

            }//if there are more than zero function calls

        }//for all the children of whenStatementDo (functionCalls)



        });
```