Implementation of a Compiler for a Stack Machine in Java

by

Anthony Y. Hui

Submitted to the Department of Electrical Engineering and Computer Science
In Partial Fulfillment of the Requirements for the Degrees of

Bachelor of Science in Computer Science and Engineering

and

Master of Engineering in Electrical Engineering and Computer Science

at the
Massachusetts Institute of Technology
February 6, 2001

© 2001 Massachusetts Institute of Technology
All rights reserved

The author hereby grants to MIT permission to reproduce and to
distribute publicly paper and electronic copies of this thesis document in whole or in part.

Signature of Author _____
Department of Electrical Engineering and Computer Science
February 6, 2001

Certified by _____
Mitchel Resnick
LEGO Papert Associate Professor of Learning Research
Thesis Supervisor

Accepted by _____
Arthur C. Smith
Chairman, Department Committee on Graduate Thesis

BARKER
MASSACHUSETTS INSTITUTE OF TECHNOLOGY
JUL 1 1 2001
LIBRARIES

Implementation of a Compiler for a Stack Machine in Java

by

Anthony Y. Hui

Submitted on February 6, 2001 to the Department of Electrical Engineering and Computer Science In Partial Fulfillment of the Requirements for the Degrees of Bachelor of Science in Computer Science and Engineering and Master of Engineering in Electrical Engineering and Computer Science

ABSTRACT

Interactive C is a programming environment that allows a user to program special controller boards used in robotics. Through a command line interface, the user can load and unload source files written in IC or binary ICB files via a serial interface between the host machine and the controller board. The user may also interactively enter code that is dynamically compiled and executed. The core of Interactive C is a special compiler that reads source files and generates stack machine code that will be executed on the virtual machine found on the controller board. The compiler is broken down into four primary modules: scanner, parser, flattener, code generator. Each module is responsible for taking the high-level C instructions and bringing them closer to the stack machine code.

Thesis Supervisor: Mitchel Resnick

Title: LEGO Papert Associate Professor of Learning Research

Acknowledgments

Special thanks to Dr. Fred Martin for giving me the opportunity to work on this project and providing much help and guidance in putting this program together.

Special thanks to the MIT Media Lab for funding this project.

Chapter 1

Introduction

1.1 Description of Interactive C

Interactive C is a special programming environment that enables a user to interface with special robotics controller boards such as the MIT 6.270 rev 2.2.1 Robot Controller Board and the Handy Board. Interactive C is executed on a host machine that connects to the controller board via a serial connection. A command line interface allows the user to compile and load code to the controller board to be executed at a later time.

User source code is written in a special variant of C known as IC that supports a subset of the C constructs and supports additional special functions. The user source code is found in .c files that can be loaded to or unloaded from the controller board where the code is executed.

In addition to IC source code, Interactive C also supports ICB and LIS files. ICB files are special binary files that are created from assembly files. These files may contain global variables and functions that may be accessed by other C files. All ICB global variables are integers and all ICB functions have a single integer argument and return an integer. LIS files contain a list of C and ICB filenames. Each filename is found on its

own line. LIS files allow a user to give multiple filenames to Interactive C to be loaded at once.

In addition to compiling and loading code, Interactive C also allows a user to reload or unload files. Interactive C also contains a feature known as interactive mode. This feature allows the user to dynamically compile, download and execute code. At the same time, any functions and global variables from previously loaded files can be accessed. This feature is very useful for debugging problems in the robots that these controller boards are found in.

The compiled code that is generated by Interactive C is not native assembly code. Instead it is pseudocode that is executed on a virtual stack machine found on the controller board. The stack machine interprets the pseudocode generated by Interactive C in order to execute the program.

1.2 Motivation for Re-implementing Interactive C in Java

This thesis project focused on re-implementing Interactive C in Java. While the current implementation of Interactive C is functioning, it was written in C and has not been updated for several years.

Re-implementing Interactive C in Java opens the door for future improvements in the hardware that Interactive C can support. The two most popular platforms for running Interactive C, the MIT 6.270 rev 2.2.1 Robot Controller Board and the Handy Board, are based on the Motorola 68HC11 platform with a processor clock speed of only 2 MHz.

Re-implementing the program also allows for improvements to what Interactive C can do. For example, the current version of Interactive C doesn't give detailed error messages and often lacks error recovery while compiling.

Having Interactive C written in Java will also allow Interactive C to be able to run on any platform that supports Java. This allows the program to be able to run almost anywhere and eliminates compatibility issues.

Chapter 2

Design Overview

2.1 Design Requirements

The design of the new version of Interactive C was largely based on what is supported by the existing implementation of Interactive C. This includes:

- Compilation of C, ICB, & LIS files
- Support for interactive mode
- Support for interfacing with the controller board through a serial connection

The selection of what constructs to support in the language definition was also based largely on what was supported by the existing implementation. To maximize backward compatibility, the language definition includes as much of what is currently supported[1].

2.2 General Description of Design

The new implementation of Interactive C was designed around a set of modules. A modular design allowed for easier development and allows future changes to have a

[1] See Appendix A for full language definition.

lesser impact on the rest of the program. The program was divided into three primary modules:

- *Main Module:* The main module is the central core of the program. It contains the user interface and a file management system that keeps track of what files have been loaded. The main program is responsible for calling the compiler and board modules at the appropriate times.

- *Compiler Module:* The compiler module contains all the functionality that compiles C, ICB, and LIS files into pseudocode. Since compilation is a complicated task, the compiler module was divided into several sub-modules:
  - Scanner: tokenizes the input file
  - Parser: parses the tokens and generates a parse tree
  - Flattener: flattens the parse tree
  - Code generator: generates the final pseudocode

- *Board Module:* The board module contains all the functionality for interfacing with the controller board via the serial port.

In subsequent chapters, a detailed description of how each module is designed will be given. Chapter 3 will cover the main module, chapter 4 will cover the compiler module, and chapter 5 will cover the board module.

Chapter 3

Main Module

The main module is the central part of the program. Its primary purpose is to control interaction with the user and keep track of important information from files that are already loaded. The main module (and Interactive C) can be executed by running "java ic" from the directory where the *ic.class* file is found.

3.1 User Interface

The user interface is the key component of the main module. It consists of a text-based command line interface. Users type in commands through that interface to access and run all available functionality of Interactive C. There are a finite number of special commands that perform special tasks. These commands are:

- *load <filename>:* loads the specified file, compiles this file and any other previously loaded files and downloads the new code to the controller board. The loading of a file can be thought of as the "checking in" of the contents of that file. The global variables and functions found in that file are now available and can be referenced by other files.

- *unload <filename>:* unloads the specified file, recompiles the remaining loaded files and downloads the new code to the controller board. The unloading of a file can be

thought of as the "checking out" of the contents of that file. The global variables and functions of that file are no longer available. Thus, if some other file references one of those variables or functions, an error is returned.

- *debug:* toggles debug mode. If debug mode is activated, debug printouts are made.

- *list files:* lists all files that are loaded. If a LIS file was loaded, the contents of the LIS file will be displayed rather than the name of the LIS file.

- *list functions:* lists all functions from files that are loaded.

- *list globals:* lists all global variables from files that are loaded.

- *list defines:* lists are #define variables from files that are loaded.

- *exit/quit:* exits from Interactive C.

- *help:* displays a description of all commands available to Interactive C.

Anything else that is inputted is considered to be interactive code to be executed on the controller board. In addition, when starting up the program, the user may elect to run Interactive C in simulation mode. Simulation mode operates without making any calls to the board module. Code is only compiled and not loaded. This is useful when running Interactive C without the controller board. Adding "-sim" to the command used to start Interactive C will activate simulation mode.

3.2 Information Stored in the Main Module

Between commands specified by the user, certain information needs to be kept. In particular, files that are loaded will require that particular pieces of information about the contents of that file be retained so that they can be used when compiling other files. For example, other files may access the global variables of another file and are required to have knowledge of those variables.

The storing of information was implemented to improve the performance of the program. An ad hoc method of handling multiple loaded files would simply involve fully recompiling all the loaded files every time a file is loaded. While this would work, it would be very inefficient since we would be constantly reanalyzing files that have been analyzed before.

Instead, several pieces of information about each file are retained. This information consists of:

- *A Program Signature:* A *program signature* for a file consists of information that is useful to other files. This includes:
  - A table listing information about the functions of that file
  - A table listing all the global variables
  - A listing of all the #define variables
  - Information about ICB binary code if the *program signature* belongs to an ICB file

- *A Byte Array Representation of the File:* This is kept so that the file does not need to be read in again in the event we need to reanalyze the file. In fact, when a file is loaded, changes made to the actual file are not visible to Interactive C until the file is loaded again.

In addition, when code is entered interactively, it needs to directly access compiled code. Thus, the following information about that last compiled code is also kept:

- The addresses where each global variable is found
- The addresses corresponding to labels to locate where functions can be found

This stored information is passed to the compiler module when it is needed and updated information is passed back from the compiler module after successful compiles.

3.3 Loading Files

When a user elects to load a file, the main module first tries to determine what type of file is being loaded. This is determined by the extension of the filename (.c, .icb, .lis). If the filename does not end in one of the recognized extensions, an error is returned.

For C files, the main module first reads in the file and saves it as a byte array. This byte array, along with the *program signatures* and byte array representations of all previously loaded files, is passed to the compiler module. If the compilation process did not return any errors, then the compiler module will return the compiled pseudocode that will be passed to the board module for loading to the controller board. Furthermore, the *program signatures* and byte array representations of the file that was just loaded are saved and the file is recorded as being loaded. In addition, the addresses where global variables are found and the addresses corresponding to functions are also saved and will be used by interactively entered code.

For ICB files, the main module only creates a pointer to the file and passes that along with the *program signatures* and byte array representations of all previously loaded files.

A byte array representation is not created for ICB files since the actual ICB file does not need to be analyzed again once it is has been analyzed once. Only C source files need to be analyzed again for the purposes of tokenizing the contents of the file. All the relevant information for an ICB file is already found in the *program signature* of that file. Once again, if the compilation succeeds, the compiled pseudocode will be returned back to the main program to be downloaded to the controller board. However, since there is no byte array representation of the file, only the *program signature* of the file, the global variables addresses and function addresses are saved.

For LIS files, the main module first parses the file and determines the names of the files that are to be loaded. If there are any errors in those filenames (i.e.: bad extension, file not found), compilation of the entire LIS file is halted. For any C files that are listed, a byte array representation of that file is created. For any ICB files that are listed, a pointer to that file is created. The collection of byte arrays and pointers to ICB files is passed to the compiler along with the *program signatures* and byte array representations of all previously loaded files. As before, if compilation succeeds, the compiled pseudocode will be returned and passed to the board module and all relevant information is updated.

3.4 Interactive Mode

When code is entered interactively, it is handed off to the compiler along with relevant information from previously loaded files. In addition, the compiler module also receives information regarding the addresses of loaded global variables and functions.

This information is needed to allow the compiler to be aware of the location of global variables and functions in the event they are referenced by the interactively entered code.

If compilation succeeds, the compiled pseudocode is passed back to be sent to the board module for downloading. In addition, the code that is entered will occasionally return information. For example, if the user inputted `"1+1;"`, then the stack machine should execute this and return 2. However, in order to be able to return this information, the main module needs to know what type of data is being returned (integer, long, float, etc.). This information is passed back from the compiler module where it is used by the main module to print out the returned value.

Chapter 4

Compiler Module

The compiler module is the most crucial part of Interactive C. It is responsible for compiling C, LIS and ICB files. This module has four separate public methods that can be called for compiling the three different types of files and handling interactively entered code.

The compiler module is broken down into four primary sub-modules: scanner, parser, flattener, and code generator. The flattener and code generator are used in all compilations. However, the scanner and parser modules are only used for compiling C files, C files found within LIS files, and interactively entered code. All ICB files and ICB files found within LIS files do not use the scanner or the parser. Instead, a special module designed specifically for handling ICB files is utilized to parse the file.

The following sections will describe the scanner, parser, special ICB parser, flattener, and the code generator in detail.

4.1 Scanner

The first module of the compiler is the scanner. The scanner's purpose is to perform lexical analysis on the input. This is done by taking the source code and recognizing reserved words, special symbols, numbers, and variable names. At the same time, white space and comments are filtered out.

This task is accomplished by constructing a deterministic finite automaton (DFA). A DFA consists of a set of states and sets of transitions between those states. Starting at a particular state, the DFA determines which transition to take based on the next character that is being read from the input file. Upon reaching particular states, that DFA will indicate that a particular token has been found.

Normally, the task of producing a scanner for a language is very tedious. However, tools are available that simplify this task by automatically generating a lexical analyzing DFA based on a special input. For the new implementation of Interactive C, a special tool called JLex was utilized.

JLex requires that the user enter a listing of all possible tokens for the language that will be recognized. In the case of the Interactive C, this included all reserved words, various special symbols, variable names, and numbers. JLex also requires that the formatting for valid comments be specified to allow it to recognize and ignore them.

Once executed, JLex will read in an input file and generate a list of tokens. However, there are situations where lexical analysis may encounter errors with the input file. The following errors are caught and reported:

- Numeric overflow: if a numeric value is detected by JLex, it will be checked to see if it is within bounds for integers, longs, and floats. If it isn't, an error will be returned.

- Variable names that start with numbers: if a variable name starts with a number, it is reported as an error back to the user.

Any other error that is encountered by JLex, such as unrecognized characters, will yield an error message indicating the line number and file name of the error. If any errors are caught during this stage of compilation, all subsequent stages are not executed since the code supplied by the user had erroneous inputs.

4.2 Parser

The parser takes the tokens that were generated by the scanner and performs syntactic and semantic analysis. Syntactic analysis involves checking that the source code follows the language specifications. Semantic analysis checks to make sure that syntactically correct source code is being used correctly according to the semantic rules of the language. For example, while `"x=a+b;"` is syntactically correct, it may not be semantically correct since "a" and "b" may be of incompatible types (i.e.: an integer and a float). When the parse stage is complete, it should generate a parse tree that serves as hierarchical intermediate representation of the user's source code. This representation is passed on to subsequent stages of the compiler.

4.2.1 CUP

Like the scanner, creating a parser from scratch is a very tedious task. However, tools are available that automatically generate parsers based on special inputs. In this case, a special tool called CUP was used.

CUP is an LALR(1) parser generator. An LALR(1) parser analyzes the list of tokens from left-to-right based on a context-free language (CFL) definition. The CFL is supplied by the user of CUP and consists of terminals and non-terminals. The terminals of the CFL consist of all possible types of tokens that could be created from the scanner. Non-terminals are specified in the parser. The terminals and non-terminals are arranged in rules where a single non-terminal is set to equal any number of terminals or non-terminals.

The CFL representation of the language is used by the LALR(1) parser to reduce the language down to a specified non-terminal known as the start symbol of the language. During this process, the parser looks one token ahead when reading in each token. The look-ahead helps the parser decide what to do with each token it is analyzing. Each token may be pushed onto a temporary stack or each token along with a certain number of tokens and non-terminals found on the top of the temporary stack may be removed and replaced by a single non-terminal based on a rule in the CFL. This process will eventually reduce all the tokens into the single non-terminal that serves as the start symbol of the entire language.

In certain instances, the CFL supplied by the user may be ambiguous. For example, when encountering the statement `"x=1+2*3;"`, the parser may not know whether to

parse 2*3 or 1+2 first. By specifying precedences in the CFL, the parser is aware of what to parse first in the event of an ambiguity. Most ambiguities occur within mathematical operations. The precedence settings of the parser follow the precedences defined by the language[2].

In certain instances, the parser encountered problems related to optional parts of a syntactic definition. For example, an if statement may contain an optional else clause. Normally, CUP would try to match as large of a string as possible. However, in the case of the if statement, CUP failed to do that. As a result, a precedence hierarchy needed to be created that gave if statements a lower priority than an if-else statement. This fix solved that problem.

4.2.2 Breakdown of Parser

For simple languages, a single parse stage is often sufficient for analyzing a language. A single parse would involve reading in the tokens from the scanner only once. However, for this compiler, the parse stage was divided into two stages.

The first parse stage is responsible for the following:

- *Collecting global variables:* A table of all global variables is built. The table is a hashtable where the name of the variable references a data structure which holds information about the type of the variable (i.e.: integer, float, etc.)

- *Collecting #define variables:* A list of #define variables is also collected. A #define variable is a variable that is associated with a literal value. The variable is treated as a global variable whose value cannot be changed.

[2] See Appendix A for precedence definitions.

Information about this variable is entered into the table of global variables and the name of this variable is entered into a list of #define variables.

- *Collecting function signatures:* A *function signature* is defined to be information that is crucial for calling that function. This includes the following:
  - The return type of the function (i.e.: void, int)
  - The name of the function
  - The types of the arguments (i.e.: an int argument)

  This information is collected in a hashtable where the name of the function references the *function signature.*

The second parse stage takes the information provided by the first parse stage and does a complete a thorough syntactic analysis. This includes building the final data structure that represents the result of the parse.

The parser was broken down in two to ease its implementation. One situation where this helped was in dealing with functions. Unlike a variable, a function can be used before it is declared in the file. Thus, when a function is called, the parser may not have any information about that function since it has not been parsed. By having the two-stage parse, the first parse stage can be used to go through the entire file and record all the functions that are available so that the second parse stage can use that information to parse all function calls.

Breaking down the parser into two stages is also useful for parsing multiple files. Multiple files may need to be parsed simultaneously when compiling a LIS file. Since files within the LIS file may reference the global variables and functions found in another file within the LIS file, it is necessary for the parser to extract all the functions and global variables of all the files before proceeding. The first parse stage performs this task.

4.2.3 Data Structures

Both parse stages generate information that is used by the compiler. To store this information, specially customized data structures were created. For the first parse stage, a *program signature* is created for each file. The *program signature* is exactly the same as the one described in the main module.

For the second parse stage, a complete hierarchical data structure representing all files in the program needs to be built. This structure is known as a parse tree and is the final result of the parse stage. To describe the data structure that is created, the following will explain the major components of the parse tree from the bottom to the top of the hierarchy and discuss how the final parse tree is built.

4.2.3.1 Variables

The *variable* data structure is used to represent any variable in the program. It includes the following information:

- The variable's type (i.e. int, float, etc.).
- The kind of variable. This refers to whether it is a global variable, a local variable, an argument, or a #define variable.
- The name of the variable.
- The scope of the variable which refers to the name of the function that the variable belongs to if it is not a global variable or a #define variable.
- The initial value of the variable if it is initialized.
- The array size of the variable (this is set to 0 if it is not an array).
- The array index in use if this variable is being used to reference a specific location found in an array.
- A flag indicating whether this variable originated from an ICB file.

4.2.3.2 Expressions

The *expression* data structure is used to describe a collection of terminals and non-terminals in the parser. The common trait of all expressions is that they all must have a data type associated with it (i.e.: integer, float, etc.). The following lists the different kinds of expressions that are available:

- *Literal Expression:* represents any constant. This includes:
  - Integer constants that are decimal, hexadecimal, or binary
  - Long constants that are decimal, hexadecimal, or binary
  - Float constants
  - Char constants
  - String constants

- *Location Expression:* represents any expression that consists of a variable.

- *Function Call Expression:* represents a call to a function. The expression consists of a *function call* object that contains the following information:
  - Return type of the function
  - Name of the function
  - A list of expressions that are being passed to the function as its arguments
  - A flag indicating whether the function originated from an ICB file

- *Binary Operation Expression:* represents any binary operation. Binary operations have an operation and two operands. These two operands consist of any expression including another *binary operation expression*.

- *Pre Unary Operation Expression:* represents a unary operation that is performed on an expression. There is a single operation and a single operand that may be any expression. The operation is restricted to those that appear before the operand. Examples of such operations include sin, cos, tan, and exp.

- *Cast Expression:* represents a casting of an expression. Casting refers to the conversion of an expression from one data type to another. The data type of a *cast expression* is the data type that the expression is being converted to.

4.2.3.3 Statements

The *statement* data structure represents a clause in the user's C source code that is terminated by a semicolon. Statements are built primarily from expressions but also contain variables and various operations. The following are the different kinds of statements:

- *Assignment Statement:* represents any situation where a variable is assigned to any expression. For example, "`x=1+1;`" is an *assignment statement.*

- *Function Call Statement:* represents a function call. This differs from a *function call expression* in that this is a stand-alone function call that is not used within another statement or expression. A *function call statement* consists of a *function call* object representing the call to that function.

- *If-Else Statement:* represents an if-else statement. It consists of an expression that represents the condition of the if statement and two blocks of statements. One block of statements represents the statements that are executed if the conditional expression was true. The other block of statements refers to those that are executed if the conditional expression was false.

- *While Statement:* represents a while loop. It consists of a single conditional expression and a block of statements that represents the code that will be executed during each iteration of the loop.

- *Printf Statement:* represents the printf command. It consists of the string that will be printed out and a list of arguments that may be included in the string being printed out.

- *Break Statement:* represents the break command to break out of a loop.

- *Return Statement:* represents the return command. It contains the expression that is returned if that is specified in the code.

- *For Statement:* represents a for loop. It contains a list of statements representing the optional initial statements executed before entering the loop, the single expression that determines whether to continue with the loop, an optional list of statements representing what is executed after each iteration of the loop, and a list of statements consisting of the code that is executed during each iteration of the loop.

- *Post Unary Operation Statement:* represents a single unary operation that is used after a variable. An example of this is "++." These operations can only be applied to variables.

- *Arithmetic Assignment Operation Statement:* represents an operation on a variable and a single expression. An example of this is the "+=" operation. This type of statement contains the variable that is being operated on, the operation that is being performed, and the expression that is the operand for the operation.

4.2.3.4 Functions

A *function* is a data type that contains all the information related to a function. This includes all the information found in a *function signature* and the following:

- An arguments hashtable where the name of the argument references a *variable* object that represents that argument.

- A list of *variable* objects representing the local variables of that function
- A hashtable of local variable names referencing *variable* objects representing local variables of this function
- A list of statements that represents the statements found within the function.

4.2.3.5 Program

The top level of the parse tree is known as a program. A *program* contains all the information found in a *program signature*. In addition, it contains a list of functions. Normally, a *program* object is used to represent the contents of a single file. However, it can be adapted to represent the contents of multiple files including ICB files. This is since the *program* object consists of lists and hashtables whose size is unlimited. Several files can be represented within a single *program* data structure by simply merging the contents together.

4.2.3.6 Building the Parse Tree

The parse tree is built by first merging the *program signatures* of all previously loaded files into a single *program signature* object. This is done right before the compiler enters the parse stage. After the first parse stage, the *program signatures* of all the files that are being loaded are merged into the combined *program signature* representing all the files. This is then passed to the second parse stage which uses that information to build a single *program* object that represents all the files that have been previously loaded and all the files that are being loaded.

The *program* object is the final output of the parse stage and is an intermediate representation of the program. An intermediate representation serves as a lower-level

representation of the program that is closer to the final compiled code. Intermediate representations are useful steps in compilation. The ultimate goal in compiling is to translate high-level code into low-level code. An intermediate representation serves as an intermediate step in this process and allows for easier debugging between modules.

4.2.4 Semantic Checks

Embedded within both stages of the parser are checks for semantic correctness. These checks verify that the syntax of the language is being used correctly. If any error is found, an error message is returned to the user. The message consists of the name of the file where the problem occurred, the line number of the error, and a detailed message describing the error.

Whenever possible, it is optimal to continue parsing the file after an error is found. A file may potentially have numerous errors. Halting the parser on each error will force the user to recompile repeatedly before all semantic errors are caught. This process of allowing compilation to continue after errors is known as error recovery. For all semantic checks, features for error recovery are included to insure that parsing continues.

One significant error recovery feature was the creation of the "error" data type. If an error occurs that affects an expression, the data type that the user intended to use is uncertain. Thus, instead of using one of the normal data types, the "error" data type is given to that expression. Any semantic check for correct data types will not be utilized if any of the expressions being analyzed has the "error" data type.

Both parse stages have semantic checks embedded within them. If there are any errors in the first parse stage, the second parse stage and all subsequent stages of the compiler will not run. Similarly, if any errors are found in the second parse stage, then all stages after the second parse stage will not be executed. This was done since there is no reason to proceed to subsequent stages of compiling if the code supplied by the user was not semantically correct and not used correctly in accordance with the rules of the language.

4.2.4.1 Details of Semantic Checks

In the first parse stage, the following semantic checks are performed:

- *Duplicate variable:* if a duplicate global variable declaration is found, an error is returned and the new variable declaration is ignored. Duplicate global variables also take #define variables into account.

- *Duplicate function names:* if a duplicate function name is found, an error is returned and the duplicate function is ignored.

- *Array declaration without size specified:* if an array declaration does not include the size of the array, an error is returned and a size of 1 is given as the size of the array. This check does not check to see that the array size given is an actual number since that information is not available in the first parse stage to improve the performance of the first parse stage. The initial size of the array is necessary in order to determine how much space to allocate.

During the second parse stage, the following semantic errors are caught:

- *Array index must be of integer or char type:* otherwise, a value of 2 is given to the array index.

- *Array size in an array declaration must be an integer or a char literal:* otherwise, an error will be returned and a value of 2 is given as the size of the array.

- *Initial values for a variable declaration must be a literal expression:* all initial values for all variable declarations must be literal values. This includes global variables, local variables, and all array lists. Otherwise, an error will be returned and an empty literal expression is put in its place.

- *Initial values for an array must be less than or equal to indicated array size:* otherwise, an error is returned. However, the initial value is still recognized and kept. This semantic check also applies to character arrays (strings).

- *Initial values for variables must be of compatible type:* the initial value of a variable must be compatible with the declared data type of the variable. The only special case is for char's and integers. A char may be used in the place of an integer and an integer may be used in the place of a char.

- *Duplicate variable names within the same scope:* since Interactive C is not an object oriented language, there are only three scopes for variables: local, argument, and global. Variables sharing the same name cannot be in the same scope or an error will be returned and the duplicate variable is ignored. However, if variables with duplicate names are found that are in different scopes, the variable with the higher scope is taken (global being the highest, then argument, then local). An exception to this is #define variables. No variables with the same name as a #define variable can be defined anywhere else.

- *Variables and functions must be declared:* all variables must be declared before being used. Functions must be declared, but not necessarily before the place where they are used.

- *Can only cast to and from a float:* this check is necessary because of the limitations of the final pseudocode. If this error is caught, the cast expression is still created, but the error is reported to the user.

- *exp, exp10, sin, cos, tan, atan, log, log10, sqrt operations must be performed on floats:* this check is necessary because of the limitations of the final pseudocode. If this error is caught, the expression is still created but given the error type.

- *mod, bitnot, lognot, logor, logand, bitxor, bitor, bitand, >>, << operations must be performed on integers:* if this error is caught, the expression is still created but given the error type.

- *+, -, *, /, ==, !=, <, >, <=, >= operations must be performed on compatible types:* the operands of these operations must be of compatible type. The only special case is for char's and integers. A char may be used in the place of an integer and an integer may be used in the place of a char. If this error is caught, the expression representing the operation is still created, but is set to be of the special error type.

- */ operation cannot be used on long's:* this is due to a limitation of the final pseudocode where division for long's was not implemented. If this error is caught, the expression representing the operation is still created, but is set to be of the special error type.

- *Using a variable that has not been declared:* normally, the parser would access the data structure representing this variable. However, if the variable

has not been declared, a dummy variable given the special error type will be put in its place.

- *#define variables cannot be modified:* the value of #define variables cannot be modified. If this error is caught, the statement is still kept, but the user will be alerted.

- *Return expression must be of compatible type:* the type of the expression that is returned must match the type that is specified for that function. As before, char's may be used in the place of integers and integers may be used in the place of char's. If this error is encountered, the statement is kept, but the user will be alerted of the error.

- *Number of arguments in a printf statement must correspond to what is indicated:* in a printf statement, arguments may be specified in the string. The number of arguments that are supplied must equal the number of arguments specified by the string supplied to the printf statement. If this error is encountered, the statement is kept, but the user will be alerted of the error.

- *Types of arguments in printf statement must correspond to what is indicated:* in a printf statement, arguments may be specified in the string. The argument specifications are by data type. The data type of each argument must correspond to what is indicated in the string. As before, char's may be used in the place of integers and integers may be used in the place of char's. If this error is encountered, the statement is kept, but the user will be alerted of the error.

- *#define variables cannot be associated with non-char arrays:* if this error is caught, a dummy variable with an error type is put in its place.

While all of the semantic checks described above take place within the parser, one of the semantic checks takes place outside of the parser. This check verifies that a function that is declared to return something actually has a return statement under all possible paths of execution. Thus, if the last statement of the function is an if-else statement, then both the if-block and the else-block needs a return statement.

While this check can be done within the parser, it is much easier to run this check right after the parser is complete when a complete parse tree can be analyzed. If this error is caught, then an error message is returned indicating the name of the function where the error occurred.

4.3 ICB Parser

The ICB parser is a special module that is used only when compiling ICB files. The module reads in the file and looks for specific components of the file that are necessary for the compiler.[3]

4.3.1 Data Blocks

The module first tries to look for the data blocks. This is done by reading in each row and determining if that row starts with the "S1" flag denoting a line of data. If a line of data is detected, the module breaks down that line into its appropriate components.

[3] See Appendix B for proper formatting of ICB files.

For the first data block, the address where the code is built is noted and the binary code is saved. For the second data block, the binary code in there is read and compared with the binary code from the first block. If differences between the two blocks are found, the location of the differences is recorded as an address location that needs to adjusted when the final code is loaded.

4.3.2 Variables and Functions

Once the data blocks have been parsed, this module then looks for global variables and functions in the file. Variables and functions can simply be found by looking for lines that start with the "variable_" string and the "subroutine_" string respectively. Once they are found, their names are recorded into a special hashtable that stores the address where the variable or function can be found.

The address is adjusted to be the address that is used if the variable were at the address where the first data block was built. The address that is supplied in the ICB file is with respect to the address used to build the second data block.

4.3.3 Error Checks

While reading in the data blocks, certain errors in the ICB files are checked. If these errors are caught, the ICB parser will return an error message indicating the file where the error occurred, the line number where the error occurred, and a description of the error. The errors that are caught are:

- *Improper numeric format:* Numbers in an ICB file are in hexadecimal. For portions of the code that require a numeric value (byte count, checksum, start address, etc.), a non-hexadecimal character will trigger an error.

- *Length of data:* If the actual amount of data in a line does not match what was indicated, an error is returned.

- *Checksum error:* If the checksum procedure fails to yield the correct result, an error is returned.

- *Data blocks must be consecutive:* The two data blocks must occur consecutively in accordance with the proper formatting for ICB files.

- *Duplicate variable/function names:* If any previously loaded file or the current file has variables or functions that share the same name, then an error will be returned.

4.3.4 Integrating with the Rest of the Compiler

When an ICB file is being compiled by itself, the ICB parser is the first part of the compiler that will handle the file. The results of the ICB parser are stored in a *program signature* object that includes all relevant information from all previously loaded files. Within the *program signature* object are fields that specifically store the binary code, the locations where addresses are located and need to be adjusted, a list of ICB global variables, and a list of ICB functions.

Once this is complete, the compiler will proceed to the stage two parse to parse all previously loaded C files and rebuild the parse tree. This step does not affect the ICB code but is necessary for previously loaded C files.

If the ICB file is found within a LIS file, the compiler will perform a similar task of having the ICB parser parse the file. However, it will also allow all other C files to go through the scanner and the stage one parse before going into the stage two parse. This is necessary since the ICB parse module extracts all the global variables and functions like the stage one parse. Other files may reference these variables and functions and will need to know that they exist before we can proceed with the stage two parse.

When multiple ICB files have been loaded, information relevant to each file cannot simply be merged together in the program object. For example, the lists of address locations needs to be separated for each file so that the compiler is aware of the specific file the address location is referring to. Thus, instead of simply storing the information as one big combined list, ICB information is stored as a list of lists. For example, the 4^{th} item in each list refers to information relevant to the 4^{th} ICB file that was loaded.

4.4 Flattener

Following the parser is the flattening stage. As is evident by looking at the structure of the parse tree, the result of the parser is a very hierarchical structure. This contrasts the linear and sequential structure of the final pseudocode that is generated by the compiler. While generating the pseudocode directly from the parse tree is possible, the intermediate task of flattening the parse tree was significant enough to justify the creation

of this module to perform this task and create a separate intermediate representation of the file.

4.4.1 Data Structures Used in Flattener

Like the parser, the flattener generates an intermediate representation of the code that requires a data structure to hold it. The data structure is known as a *flat program*. The *flat program* object consists of the following:

- A list of global variable names
- A hashtable where each function name references a list of *variable* objects that represents the arguments of that function
- A hashtable where each function name references a list of *variable* objects that represents the local variables of that function
- A list of listings of ICB binary code
- A list of listings of address locations in the ICB binary code that need to be adjusted in accordance with the place where the binary code is loaded to
- A list of listings of global variables found within each ICB file
- A list of listings of functions found within each ICB file
- A list of *instruction* objects

The list of *instruction* objects is the primary means of representing the program in the *flat program* object. In the *program* object, the representation of the program was hierarchical. However, for the *flat program*, the program is represented as a single list of instructions.

Statements within the parse tree are flattened into one or more instructions. Often, a single statement results in multiple instructions since statements are often complex and

cannot be performed by a single instruction. For example, `"x=1+2+3;"` cannot be performed in a single operation since, as evident when looking at the available pseudocode commands, adding can only be done on two operands at a time.

Thus, when breaking down these statements, the flattener will need to keep track of numerous intermediate steps. To do this, every intermediate step is represented by a system variable. A system variable uses the data structure of a variable. The only difference is that the variable is assigned the special "system" data type to indicate that this variable was created by the flatten stage and is not a variable designated by the user's source code.

Going back to the `"x=1+2+3;"` example, this statement would be broken down into:

```
temp=1+2;
x=temp+3;
```

"Temp" is a system variable used to represent the intermediate steps of this complex operation. In the flattener, all system variables are give the name "$sysVar#" where "#" is a number. A counter keeps track of the number to assign to prevent duplicate system variables from being assigned. The "$" is the first character of the name of the system variable since a variable in the user's source code cannot start with a "$." Thus, this prevents a system variable from sharing the same name as a regular variable defined by the user.

In addition to the situation just described, system variables are also employed throughout the flattening process in other circumstances. In general, a system variable is used to represent the writing to a temporary memory location in the final pseudocode. In particular, the pseudocode that gets generated is for a stack machine where all operations

are performed on a stack. Any operation will require that the information be saved onto the stack before the operation can be informed.

Thus, for all instructions, the operands within the instruction consist primarily of variables. Thus, any non-variable object found within a statement needs to be assigned to a system variable before it can be operated on. The following is a list of all the available instructions:

- *Label Instruction:* This instruction is used to represent a marker within the list of instructions that other instructions can refer to. It contains a string that represents the name of the label. The following is the convention for label names:
  - All labels marking locations relevant to the entire program start with the "$program_" string. For example, the label marking the location where global variables are declared is labeled "$program_globals."
  - All labels marking locations relevant to functions start with the "$function_" string. For example, the start location of the "main" function is labeled "$function_main."
  - All other labels start with the "$label_" string and are followed by a number. The number is read from a counter that is incremented every time a label of this type is created. This prevents duplicate labels from being created. Additional text may follow the number in the label depending on how the label is being used.

- *Global Variable Declaration Instruction:* This represents the declaration of a global variable. It contains a field that holds a variable that represents the global variable being declared.

- *Local Variable Declaration Instruction:* This represents the declaration of a local variable within a function. It contains a field that holds a variable that represents the local variable being declared.

- *Assign Literal Instruction:* This represents the setting of a variable to a literal expression. An example would be `"x=4;"`. This instruction contains a variable and a field to hold the literal expression.

- *Assign Location Instruction 1 & Assign Location Instruction:* These two instructions are used to represent the setting of the contents of one variable to that of another variable. Two instructions are needed to represent this because of the way the pseudocode is constructed in the code generation stage. The instructions contain the two variables that are involved in this assignment.

- *Assign Array Location Instruction 1, Assign Array Location Instruction 2 & Assign to Array Instruction:* This represents assigning the contents of a variable to an element found in an array. Three instructions are needed to represent this because of the way the pseudocode is constructed in the code generation stage. The instructions contain the two variables that are involved in this assignment.

- *Array Pointer Instruction:* This represents the copying of the contents of an entire array to another array.

- *Assign Binary Operation Instruction:* This represents a binary operation being performed on two variables with its result being assigned to another variable. This instruction contains three variables representing the two operands and the variable where the result is being stored. In addition, it contains the operation that is being performed.

- *Assign Pre-Unary Operation Instruction:* This represents a unary operation being performed on the variable where the operation came before the operand in the original C code. This instruction contains a variable representing the operand, a variable representing the location where the result is being set to and the operation being performed.

- *Assign Function Call Instruction:* This represents the setting of a variable to the result of a function call. This instruction contains the variable where the result is being set to and a *function call* object representing the function that is being called.

- *Function Call Begin Instruction:* This represents the start of a call to a function. It contains the name of the label where the function is to be located based on the label convention described earlier, the name of the function, the return type of the function and a flag indicating whether the function is an ICB function.

- *Function Call Statement Instruction:* This represents the calling of a function where the function call was a stand-alone statement in the original C code (the function call was not being used as part of a statement or expression). This instruction must be preceded by a *function call begin instruction.* It contains a function call object representing the function that is being called.

- *Function Call Expression Instruction:* This represents the calling of a function where the function call was an expression in the original C code (the function call was being used as part of a statement or expression). This instruction must be preceded by a *function call begin instruction.* It contains a *function call* object representing the function that is being called.

- *Branch True Instruction:* This instruction represents a branch to a label instruction if the variable that is given is true. This instruction contains a variable that represents the condition of the branch and the name of the label to branch to.

- *Branch Push True Instruction:* This instruction represents a branch to a label instruction if the variable that is given is true. This instruction contains a variable that represents the condition of the branch and the name of the label to branch to. This instruction is used only with the "||" operation and is needed because of the way code generation generates the pseudocode for the "||" operation.

- *Branch False Instruction:* This instruction represents a branch to a label instruction if the variable that is given is true. This instruction contains a variable that represents the condition of the branch and the name of the label to branch to.

- *Branch Push False Instruction:* This instruction represents a branch to a label instruction if the variable that is given is true. This instruction contains a variable that represents the condition of the branch and the name of the label to branch to. This instruction is used only with the "&&" operation and is needed because of the way code generation generates the pseudocode for the "&&" operation.

- *Jump Instruction:* This instruction indicates a jump to a label instruction. This is similar to any of the branch instructions. However, the jump takes place under all circumstances unlike a branch which relies on a condition. This instruction contains the name of the label to jump to.

- *Printf Instruction:* This instruction represents a printf statement. It contains a field indicating the space needed for the arguments of the printf statement. This is required due to the way the pseudocode is constructed in the code generation.

- *Return Instruction:* This instruction represents a return statement. This instruction contains a variable representing what is being returned if something is being returned.

- *Function Start Instruction:* This instruction indicates the start of a function and contains the name of the function.

- *Function End Instruction:* This instruction indicates the end of a function.

- *Cast Instruction:* This instruction represents a casting operation. It contains the original data type of the expression that is being casted and the data type that the expression is being casted to.

- *Logidn Instruction:* This is a special instruction used specifically for the "||" and "&&" operations. This instruction is needed because of the way code generation generates pseudocode for the "||" and "&&" operations. This instruction corresponds to the LOGIDN command in pseudocode.[4]

- *String Literal Instruction:* This is a special instruction used only to handle string literals.

4.4.2 Details of Flattening Process

The flattener is designed to generate an intermediate structure that closely resembles the final pseudocode that is generated by the compiler. Thus, the order of the instructions is dictated by the proper formatting of the final pseudocode.

The first thing that is dealt with in the final pseudocode is global variables. Thus, the flattener will add instructions for global variables first. It adds a label instruction indicating the start of global variable handling, reads in the list of global variables from the program object, and adds a global variable declaration instruction for each global variable that is found.

Next, the flattener needs to flatten all the functions. For each function, the flattener first appends a function start instruction to indicate the start of a function. It then

[4] To see how this instruction is used, look at the description into how binary operation expressions are flattened.

accesses the local variables of the function and appends a local variable declaration instruction to the instruction list for each local variable in the function and adds the local variable to the hashtable of local variables. Similarly, arguments of the function are read in and saved to the hashtable of arguments.

Next, the flattener begins flattening the statements within the function. Each statement and the expressions within the statement will yield a specific set of instructions. A method was created that flattened all types of statements and a method was created to flatten all types of expressions. The following describes how each type of statement is flattened:

- *Assignment Statement:* If the expression is being assigned to an element in an array, then *assign array location instruction 1, 2,* and the *assign to array instruction* are used. In the process, the expression representing the index of the array we are writing to is passed to the expression flattener between the first and second instructions and the expression representing what we are assigning to the array is passed to the expression flattener between the second and third instructions. If the expression is being assigned to a non-array, then *assign location instruction 1* and *assign location instruction* are used. In between the first and second instructions, the expression representing what is being assigned to the variable is passed to the expression flattener for flattening.

- *Function Call Statement:* This is broken down into a *function call begin instruction* and a *function call statement instruction.* In between the two instructions, all of the arguments of the function are passed to the expression flattener to be flattened.

- *If Else Statement:* First, the expression representing the condition of the statement is passed to the expression flattener. Then a *branch false instruction* is used to

represent a branch to the else block if the condition was false. The statements within the if block and the else block are then passed to the statement flattener to be flattened. In between the if block and the else block, *label instructions* are placed to mark the beginning and end of each block.

- *While Statement:* First, the expression representing the condition of the statement is passed to the expression flattener. Then a *branch false instruction* is used to represent a branch to end the looping if the condition was false. The statements within the block are then passed to the statement flattener to be flattened. *Label instructions* are placed to mark the beginning and end of the block and a *jump instruction* is placed after the block of statements to represent the looping process of the while statement.

- *For Statement:* A for statement consists of two sets of statements, an expression, and a block of statements that are to be looped through. Of the two sets of statements, one of them is meant to be executed once before the start of the for loop. This set of statements is first passed to the statement flattener for flattening. Next, the expression representing the condition of the for loop is passed to the expression flattener. Then, the block of statements is passed to the statement flattener. Lastly, the set of statements that was meant to be executed at the end of every iteration is then passed to the statement flattener. In between each of the parts of the for statement, *label instructions* are placed that keep track of where specific parts of the for statement begin and end. In addition *jump* and *branch instructions* are used for flow control around the various parts of the for loop.

- *Printf Statement:* First, the string in the printf statement is passed to the expression flattener. Then, each of the arguments of the printf statement is passed to the expression flattener. Once that is completed, a *printf instruction* is added to the list of instructions

- *Return Statement:* The expression that is being returned is sent to the expression flattener and a *return instruction* is added to the list of instructions.

- *Post Unary Operation Statement & Arithmetic Assignment Operation Statements:* If the result is being assigned to an array, then the *assign array location 1* and *2* instructions are used and the index that is being used is passed to the expression flattener between the first and second instructions. Otherwise, the *assign location 1* instruction is used. This is followed by the flattening of the expression being operated on. Once that is completed, an *assign binary operation instruction* expression is added. The unary operation is translated to an equivalent binary operation (i.e.: "++" is turned into adding 1 to the expression). Lastly, an *assign location instruction* is added on if the result is assigned to a non-array and the *assign to array instruction* is added on if the result is being assigned to an array.

The following describes how the different types of expressions are flattened:

- *Location Expression:* If the location is an array, then the *assign array location 1* and *2 instructions* are used and the expression that indicates the index of the array being used is passed the expression flattener in between those two instructions. Otherwise, the *assign location 1 instruction* is used. This is followed by the flattening of the expression being operated on. Lastly, an *assign location instruction* is added on if the location is a non-array and the *assign to array instruction* is added on if the location is an array.

- *Function Call Expression:* This is broken down into a *function call begin instruction* and a *function call expression instruction*. In between the two instructions, all of the arguments of the function are passed to the expression flattener to be flattened.

- *Binary Operation Expression:* For operations other than "&&" and "||," the operands are sent to the expression flattener to be flattened and an *assign binary operation instruction* is added. However, for a "&&" and "||" operation, special instructions need to be created. The reason is that all other binary operations are translated into a single equivalent command in the code generator. For example, addition of two integers becomes the ADD2 command. However, "&&" and "||" do not have direct translations. Instead, for "&&," the code that will be generated by the code generator looks at the first operand. If it's false, it returns false. If it's true, the value is the value of the second operand. Similarly for "||,"the code that will be generated by the code generator looks at the first operand. If it's true, it returns true. If it's false, the value is the value of the second operand. To represent this, the flattener will first flatten the first operand and assign it to a system variable via an *assign location instruction.* Then, a *branch push true instruction* is inserted if the operation is a "||" operation and a *branch push false instruction* is inserted if the operation is a "&&" operation. Then, the flattener will flatten the second operand, assign it to a system variable via an *assign location instruction* and insert a *logidn instruction.*

- *Pre Unary Operation Expression:* The expression that is being operated on is sent to the expression flattener to be flattened and an *assign pre-unary operation instruction* is added.

- *Cast Expression:* A *cast instruction* is added to the list of instructions.

- *Non-String Literal Expression:* An *assign literal instruction* is added to the list of instructions.

- *String Literal Expression:* A string literal needs to be treated specially due to the special way it is handled in the pseudocode. When a string literal expression is encountered, a *jump instruction* is added. This is followed by a *string literal instruction* and a *label instruction* with the label that the previous *jump instruction*

jumped to. This is followed by an *assign literal instruction.* This is needed because in the final pseudocode, a string literal is stored with the instructions. This will become more evident after analyzing the code generator.

Once all the statements and expressions of a function have been flattened, the flattener will append a *function end instruction* and will then flatten any other functions that follow. Once all functions have been flattened, the flattener can save all the information to the *flat program* object. In addition, any information in the program object pertaining to ICB files needs to be copied over to the *flat program* object. Any information pertaining to ICB files does not need to be analyzed during this stage since the data structure holding ICB files is not hierarchical and does not need to be flattened. The binary code that was read in merely needs to be downloaded to the controller board.

As evident from analyzing the way the flattener deals with the program object passed from the parser, the flattener removes all hierarchy within the code and breaks it down into a linear list of instructions. The only remaining features of the old hierarchy are the labels that are used to indicate where portions of the instructions begin and end.

4.5 Code Generation

The last step of the compiler is the code generation module. This module takes in a *flat program* object generated by the flatten stage as its input and outputs the final pseudocode that can be downloaded to the controller board to be executed on the board's virtual machine.

Code generation is divided into two stages. The first stage is the primary stage where most of the code is generated. The second stage serves to fill in missing addresses. These missing addresses arise from labels whose actual address has not been determined since the code has not been fully built yet. For example, if we have a function call to a function that has not been built yet, the code generator is uncertain of the exact address location of where to jump to. The second stage of code generation will have that information available and will fill in any of the missing address locations.

4.5.1 How the Stack Machine is Organized

The pseudocode that is generated is designed to be executed on a stack machine. The stack machine for the virtual machine on the controller board consists of a single stack for each process. If multiple processes are running on the virtual machine, then each process is given a separate stack.[5] Items can be pushed and popped off any of those stacks. Items can be read from or written to anywhere on the stack by referencing the address relative to the stack pointer. The stack pointer points to the next free space on the stack.

All temporary memory space is provided on the stack except for global variables and the actual code. Global variables are stored starting at address location 0xBFC0 while the program's code is stored starting at address location 0x8000. Each address is for one byte of memory.

[5]This implementation of Interactive C will not take advantage of multiple processes. Instead, there is only one process running and one stack in use.

4.5.2 Stack Conventions

In general, operations that are performed on the stack have their arguments pushed first before the operation is performed. The operation will then pop the operands and may then push the result onto the stack.[6]

When a function is call, there are two parties involved: the caller and the callee. The caller is responsible for the following:

- Pushing the return address onto the stack
- Pushing the arguments in order onto the stack
- Jumping to the location of the function

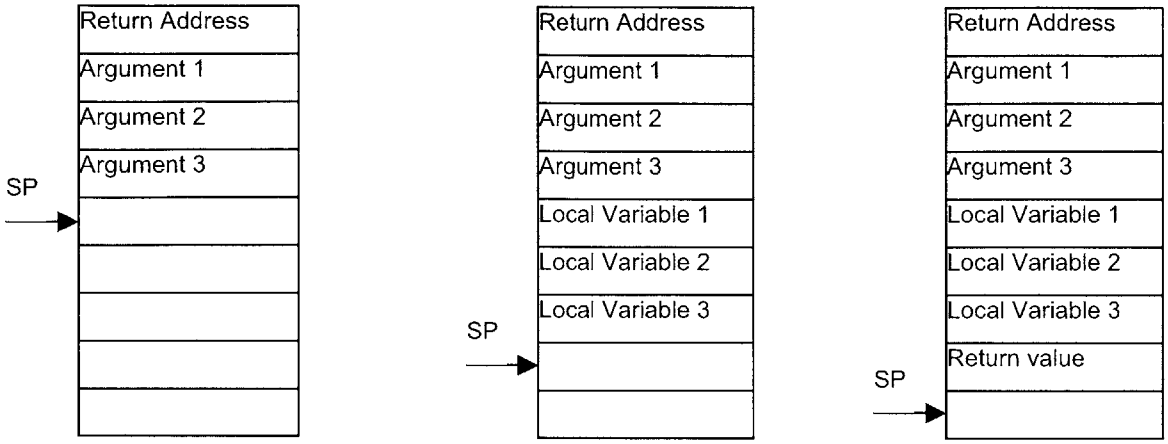
The callee is responsible for the following:

- Pushing local variables onto the stack
- Pushing the return value onto the stack (if there is one)
- Popping the local variables and anything that was pushed onto the stack while executing the program
- Popping off the arguments of the function
- Returning to the address indicated by the caller of the function

Figure 1 illustrates how the stack is used when calling a function. In step (a), the caller is about to call the function, in step (b) the callee is about to execute the code in its body, in step (c) the callee is about to return back to the caller. The pseudocode "return" instruction pops the return address from a specified address into the stack, removes the

[6] This is the general convention of the instructions in the pseudocode. For more details on how each operation works, see Appendix C.

space allocated for arguments and local variables, and restores the return value to the top of the stack when returning.



(a) Caller about to call the function

(b) Callee about to execute code

(c) Callee about to return

Figure 1

While a function is executing, temporary space will be needed to execute statements. However, the convention is that after each equivalent statement in the user's source code, the stack must clean up after itself and the stack pointer must return back to pointing at the location right after the last local variable. This effectively means that all information must be stored either as a local variable, argument, or a global variable. For example, after executing the pseudocode that is equivalent to the statement `"x=1+1;"`, the result should be stored in the location allocated for the variable "x." Any temporary space that was used should be eliminated.

The temporary space and space allocated for variables depends on the data type. The following lists the number of bytes each data type requires:

Data Type	*Number of Bytes Required*
Int	2
Char	2
Long	4
Float	4

Arrays are stored in the same locations as a regular variable. Items in the array are pushed in reverse order onto the stack. However, following all the members of the array, an integer indicating the size of the array and the current address of where the array is located is pushed onto the stack (if the array is a local variable or an argument, then the address is the current stack address, if the array is a global variable, then the address is the address where the array is being stored in the global variable space). *Figure 2* illustrates how an array with the items 1,2,3,4,5 is stored:

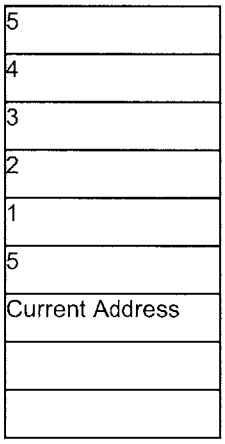


Figure 2

There is an exception to this convention for strings. Strings are not stored in the stack and are instead stored in the pseudocode's memory. When a string is used, its value is

placed in the pseudocode. A jump instruction will jump over the string constant when the code is executed.

4.5.3 Internal Information Kept During Code Generation

During code generation, a lot of information needs to be kept. This information keeps track of what has been generated and where particular items are located in memory. The following describes what information is kept:

- *Program counter:* This points to the address where the code is to be written next.

- *Stack pointer:* This points to the address of the top of the stack (the next free stack space). The actual address of the stack pointer is unknown and is not necessary since any reference to the stack will only be based on a certain offset from the stack pointer (i.e.: 2 more than the stack pointer address). Thus, knowledge of the actual address is unnecessary. The code generator starts the stack pointer address at an arbitrary value of 0 and adjusts it accordingly as items are pushed and popped from the stack.

- *A stack of stack pointer address:* Since this particular stack machine does not have a frame pointer, the exact location of the border between the elements of one function and another are unknown when executing the code. Thus, the code generator must keep track of this information so that when a function has completed its execution and is ready to return control back to the function that called it, the amount of space to free up is known. This information is kept in a stack since multiple layers of function calls, including recursive calls to the same function, may occur.

- *Hashtable of labels:* Labels are used as substitutes to address locations that may not be known when the code is generated. However, as code generation proceeds, the actual address will eventually be known and this table will map the name of the label to the actual address so that stage two of code generation can use this table to fill in the missing addresses.

- *Hashtable of global variables:* This table maps global variable names to the address where they are found.

- *Hashtable of local variables:* This table maps function names to another hashtable that maps the local variables of each function to their stack pointer location address.

- *Hashtable of arguments:* This table maps function names to another hashtable that maps the arguments of each function to the offset from the start of the local variables to this function. For example, if the function has two arguments, then the first argument will be two bytes away and the second argument will be zero bytes away from the local variables of the function (the first argument is further away since it was pushed onto the stack first).

- *Hashtable of strings:* Since strings are stored in the pseudocode instructions, the exact address location of where the string is located needs to be known. A table that maps the string to the actual address is utilized for this task.

- *Hashtable of the argument depth for each function:* This table maps each function name to the amount of space that is needed to store all the arguments of this function. This is used by a function to determine how much space to free after the call to that function is complete.

4.5.4 Format of Pseudocode

The pseudocode is designed to follow a specific format as follows:

- *Initialization code:* This is code that is executed to initialize the program. The initialization code varies depending on whether there is a function named "main." If there is a "main" function, the stack machine will execute the main function automatically. Under this case, the initial code consists of the following:
  - Pushing a 0 on the stack.
  - Pushing a 2 on the stack.
  - Pushing the address of the "main" function onto the stack (the address will initially be a label that will be filled in after stage two of code generation). This is done so that the program is aware of the location of the "main" function.
  - Pushing a 5 onto the stack.
  - Pushing a 256 onto the stack.
  - *Startprocess*
  - Popping 2 bytes off the stack
  - *Haltnotify*

  If the initial code does not contain a "main" function, then the initial code consists of the following:

  - Pushing a label "$label0"
  - Jumping to the "$program_globals" label.
  - Recording "$label0" as the value of the program counter at this point

  This code is necessary to initialize the global variables. If there is a "main" function in the program, then it is the responsibility of the "main" function to initialize all global variables before any other code is executed. However, if there is no "main" function, then the code must start out by going to the global variable initialization routines before any code is allowed to execute.

- *Global Variable Initialization Code:* This code is used to push the global variables into the appropriate location. The code is as follows:

- *Initint*
- *Sprel 0 0*
- *Push2 191 192*: this pushes the address where the global variables are to be stored
- *Setsp*
- For each global variable, its initial value is pushed onto the stack. If an initial value was not given, then a value of 0 is pushed.
- A jump to the label "$program_end." This jumps to code that is found at the end of the pseudocode that completes the initialization procedure.

- *Code for each function*: The next section consists of the code for each function. What gets produced depends on what is in the function. Each function will start with a *checkstack* command that checks that the amount of stack space required by the function is actually available. The actual amount of space required is unknown until the entire function has been built. Thus, the command takes a label as its argument. The label will be filled in during stage two of code generation.

- *Ending code*: The last section consists of code that is actually called by the initialization code. There are two different scenarios for this code. The first scenario is when there is no ICB code to load. Under this circumstance, the code is as follows:

  - *Fetchreg 0*
  - *Setsp*
  - *Mret0*

  However, if ICB code is present, then the code will be loaded in this section of code. More details about how ICB code is loaded can be found in section 4.5.6.

4.5.5 Details of How Instructions Are Translated to Pseudocode

Each instruction that is read in from the *flattened program* will be translated in a specific way into pseudocode instructions. The following describes how each instruction is translated:

- *Global Variable Declaration Instruction:* This instruction results in the pushing of the initial value of the variable into the location where global variables are stored. During the flattening process, this instruction is only generated in the beginning of the list of instructions to correspond with the fact that global variable declarations take place in the beginning of the code generation process.

- *Local Variable Declaration:* This instruction results in the pushing of the initial value of the variable onto the stack. Like the *global variable declaration instruction*, this instruction is found in the beginning of a function when the instructions are generated during the flattening stage. This corresponds to the fact that local variable declarations occur in the beginning of functions in the pseudocode.

- *String Literal Instruction:* This instruction corresponds to the usage of a string. Since strings are stored in the pseudocode's memory space, the string literal needs to be incorporated into the program. This simply involves writing each character of the string into the pseudocode's memory. Since what is found in the pseudocode's memory is normally executed, a *jumpi* instructions needs to be added that jumps over the characters of the string so that they don't get executed.

- *Cast Instruction:* This instruction is translated into one of the following pseudocode instructions:

Type of Cast	*Instruction*
Integer → Float	*int2fl*
Float → Integer	*fl2int*
Long → Float	*lng2fl*
Float → Long	*fl2lng*

These instructions will convert whatever value is on the top of the stack to the appropriate type. Adjustments to the stack pointer will need to be made when converting between an integer and a float.

- *Function Start Instruction:* This instruction results in a *checkstack* operation that verifies that enough stack space is available for this function to proceed.

- *Function End Instruction:* This instruction results in a *mret0* pseudocode instruction that takes as an argument the number of bytes to pop off the stack. This value includes all local variables and arguments.

- *Printf Instruction:* This instruction will result in a *push* operation that pushes the number of arguments in the printf statement onto the stack. This is followed by the *printf* pseudocode instruction and a pop statement that pops off the number of arguments value that was pushed on earlier.

- *Jump Instruction:* This instruction results in a *jumpi* instruction that takes in as an argument the address to jump to. Since the actual address may not be known, a label is left in its place that will be filled in during stage two of code generation.

- *Branch True Instruction:* This instruction results in a *jtrue* instruction that takes in as an argument the address to jump to. Since the actual address may not be known, a label is left in its place that will be filled in during stage two of code generation.

- *Branch False Instruction:* This instruction results in a *jfalse* instruction that takes in as an argument the address to jump to. Since the actual address may not be known, a label is left in its place that will be filled in during stage two of code generation.

- *Branch Push True Instruction:* This instruction results in a *jptrue* instruction that takes in as an argument the address to jump to. Since the actual address may not be known, a label is left in its place that will be filled in during stage two of code generation.

- *Branch Push False Instruction:* This instruction results in a *jpfalse* instruction that takes in as an argument the address to jump to. Since the actual address may not be known, a label is left in its place that will be filled in during stage two of code generation.

- *Function Call Begin Instruction:* This instruction initializes a function call and saves the value of the stack pointer. If the function being called is found in an ICB file, then the label representing the address of that function is pushed onto the stack.

- *Function Call Expression/Statement Instruction:* Both of these instructions represent calls to a function. Both instructions will result in a *jumpi* pseudocode instruction to the label that represents the address of the function. However, for a *function call statement instruction*, if the function returns something, then it needs to be popped since the value is not being used by anybody. If the function happens to be an ICB function, then a special pseudocode instruction *callm1* is used instead of the *jumpi*. *Callml* uses the address pushed when analyzing the *function call begin instruction.*

- *Return Instruction:* This instruction is specifically for situations where a value is returned. If no value is returned by a function, then the code provided after

translating the *function end instruction* will suffice. However, if a value is returned, then the value being returned is accessed and pushed onto the stack. Then, the *mret0/mret2/mret4* pseudocode instruction is added. Which of the pseudocode instructions is used depends on the data type of what is being returned.

- *Assign Binary Operation Instruction:* First, the operands for this operation have already been pushed onto the stack since the expressions representing the operands were flattened in the flattener before the statement representing this entire operation. The code generator needs to first determine the address of the variable that the result of this operation is being set to. The value of this address is pushed onto the stack. Next, the binary operation command is added to the pseudocode.

- *Logidn Instruction:* This instruction results in a *logidn* instruction.

- *Assign Pre Unary Operation Instruction:* First, the operand for this operation has already been pushed onto the stack since the expression representing the operand was flattened in the flattener before the statement representing this entire operation. The code generator needs to first determine the address of the variable that the result of this operation is being set to. The value of this address is pushed onto the stack. Next, the operation's instruction is added to the pseudocode.

- *Assign Literal Instruction:* This instruction will result in the pushing of the literal onto the stack. If the value is a negative value, then the absolute value will be pushed first, then the *neg2/neg4/fneg* operation will be applied to negate the value after it has been pushed onto the stack

- *Array Pointer Instruction:* This instruction is used when an array is accessed in its entirety rather than accessing a single element of the array. This instruction will result in the pushing of the address where the array is found onto the stack.

- *Assign Array Location 1 Instruction:* This is the first instruction in a sequence of instructions that are created in the flattener when assigning to an array. This instruction will result in a pseudocode instruction to push the address of the array.

- *Assign Array Location 2 Instruction:* This instruction will result in the pseudocode instruction *aref1/aref2/aref4* that accesses the actual address of the array item.

- *Assign to Array Location Instruction:* This instruction will result in a *poke2/poke4* instruction that assigns the value to an array.

- *Assign Location 1 Instruction:* This instruction pushes the address of the variable we are reading in onto the stack.

- *Assign Location Instruction:* This instruction will either push the value of the variable onto stack, store it in a global variable, store it in a local variable, or store it in an argument.

- *Label Instruction:* This instruction will, under most cases, result in the label being set to the current program counter in the hashtable of labels. However, there are special labels that are treated differently:

    - $program_start: This will trigger the code generator to generate code for the initialization section of the pseudocode.
    - $program_globals: This will trigger the code generator to generate code for global variable initialization section of the pseudocode.

4.5.6 Building of ICB code

As mentioned in section 4.5.4, ICB binary code is built at the end of all the pseudocode. Unlike normal user code, each ICB file is built separately in the pseudocode. For each ICB file, the relevant information is available in the *flat program* object. Most of what gets actually downloaded is already available to us. However, there are addresses within the ICB code that need to be adjusted based on the address that the ICB code is built in.

These adjustments can be made by accessing the list of locations within the ICB code where addresses are located. For each location, the value is then adjusted based on where the code provided in the ICB file was built and where the code actually gets built. For example, if the code in the file was built in 0x8000 and the code is now being built in 0x8080, then 0x80 should be added to each address found in the ICB code.

Once the adjustments are made, the code can be downloaded as follows:

- First, a *jumpi* is added. This jumps to a label that is found after the ICB code.
- The ICB code is downloaded
- A label is added to mark the spot where the jump statement in the first step will jump to. The jump is added so that when the pseudocode is executed, the binary code is not executed.

Once the code is downloaded, the compiler needs to record the addresses where functions and global variables are found. These addresses also need to be adjusted to reflect where the code is being built.

In addition, some ICB files contain a special function called "initialize_module." This function initializes the ICB code and needs to be executed immediately. Thus, if

such a function in found in an ICB file, then it has to be run and the following pseudocode is appended:

- A push command to push the address of the "initialize_module" function.
- Push 0 onto the stack
- *Callml*
- A *Pop2* command that pops off what is returned by the "initialize_module" function.

4.5.7 A Simple Example

The following is an example of how the pseudocode is generated for the statement `"x=2+3;."` First, the instructions that are generated for this statement in the flatten stage are as follows:

- Assign Location 1
- Assign Literal: Assigns "2" to a system variable
- Assign Literal: Assigns "3" to a system variable
- Assign Binary Operation: Assigns results of operation onto a system variable
- Assign Location: Assigns result of operation to variable x

These instructions are then translated to the follow pseudocode instructions

- push2 191 190: this assumes that x is global variable
- push2 0 2
- push2 0 3
- add2
- poke2

As evident in this example, when many of the instructions are given to the code generator, the instructions that were previously passed are unknown. Thus when translating certain instructions that require arguments (i.e.: Assign Binary Operation

Instruction), the code generator simply assumes that code to push the operands onto the stack has already been generated. Thus, the code generator is merely a direct translator and relies on the previous stages of the compiler to generate correct code.

4.6 Compiling Interactively Entered Code

When compiling code that is executed interactively, numerous portions of the compiler are omitted and some extra features are required. The following describes how each module of the compiler is affected:

- *Scanner:* The scanner is not affected.
- *Parser:* Normally, the parser will return a *program* object that represents the entire program. However, in the case of interactively entered code, an entire *program* is not created. Thus, three special constructs were added in stage two of the parser that accept the following:
  - Expressions
  - Statements
  - A block of statements enclosed in braces

  These special constructs are given the lowest precedence. Thus, they are only reached if all other constructs have already failed. This allows the parser to function for normal programs. However, for interactively entered code, the parser will not be able to find correct constructs for the interactively entered code since none of the code is found within a function. Thus, the only constructs that will work are the ones specified for interactively entered code.

- *Flattener:* The flattener has a special method for handling interactive code. This method only accesses portions of the flattener that generate instructions for

statements and expressions. All sections of the flattener that handle functions and initialization of global variables are not accessed.

- *Code Generation*: The code generator only generates pseudocode for statements and expressions. All the initialization code and ending code is omitted. In addition, the code generator for interactively entered code requires the address locations of global variables and functions of previously loaded files. The code generator uses this information if the interactively entered code happens to reference such information.

In addition, when code is entered interactively, all information related to previously loaded files is left unchanged.

Chapter 5

Board Module

The board module contains all the functionality that allows the compiler to interact with the controller board. The proper protocols for interacting with the controller board are based on what is done by the current implementation of Interactive C.

The board module has two different ways to download code. One method is for loading files while the other is for loading and executing interactively entered code. Both methods take in a listing of the compiled pseudocode as its input.

5.1 Details of Board Downloading

The protocol for loading interactively code uses a subset of the methods used in loading code. To download interactively entered code, the following protocol is used:

- The code is downloaded to the controller board in 100 byte blocks starting at address 0xC200. This address marks the beginning of space that is used by the controller board to run interactively entered code.
- Writes the constant 186 directly after the spot where the code is downloaded to. 186 represents the *haltnotify* pseudocode command.
- Reads and saves initial location of the stack pointer from address 0xC302. 0xC302 is the designated address where the value of the stack pointer is stored.

- Writes the address 0xC200 to the process buffer (0xC300).
- Clears memory at location 0x0B.
- Writes 0 to location 0x309. Location 0x309 is where the status of the process being executed is stored. Setting it to 0 resets the value.
- Read in the value at memory location 0x0B
- Read in the final stack pointer value.
- Read in the value located at the final stack pointer. This contains the value being returned for interactively entered code.
- Save the initial stack pointer back to location 0xC302.

The protocol for loading code uses the protocol described above. However, additional steps are taken before running the above protocol:

- Kill all processes
- Kill all interrupts
- Download code to the controller board in 100 byte blocks starting at address 0x8000.

Once these steps are completed, the board module will proceed with the same steps that are used to download interactively entered code. However, the code that is downloaded during this step jumps to the location where the code is initialized. This will allow the initialization routines to be executed.

The user is mostly kept away from any of the details of the board module. However, the user is informed of the size of the file being loaded and the address location where the code is going to.

Chapter 6

Conclusions

The new implementation of Interactive C satisfies the requirements of compiling C files, compiling ICB files, compiling LIS files, supporting interactive mode, and interfacing with the controller board through a serial connection. The program was divided into modules to ease implementation and facilitate future changes made to the code. While the program performs the functionality of the current implementation of Interactive C, it also provides improved handling of errors and improved portability through the use of Java. In addition, the new implementation lays the groundwork for improvements in the hardware that Interactive C can support.

While the program is useable at its current state, there is much room for improvement. The following are items where this program can be improved:

- *Downloading Speeds:* Unfortunately, the Java package for interfacing with the serial port (javax.comm) is very slow. As a result, the amount of time required to execute the board module every time a file is loaded is excessively long. Time should be invested in finding a speedier way of downloading code.

- *Optimizations:* While test trials have found that the code is relatively optimal, an investigation should be done into whether significant optimization algorithms

should be incorporated. These optimizations may include dead-code elimination, mathematical simplifications, and loop unrolling.

- *Additional Language Support:* Additional language support should be looked into to expand the language the user can utilize. For example, support for multi-dimensional arrays may be added in the future.

Appendix A

Language Definitions

A.1 Lexical Considerations

All identifiers and reserved words are case sensitive. The reserved words are:

#define, atan, break, char, const, cos, else, exp10, exp, float, for, if, int, log, log10, long, null, printf, return, sin, sqrt, tan, void, while

Comments can be stated by either the following two methods:

- Start with // and terminate with the end of the line
- Start with /* and terminate with */

White space (one or more spaces, tabs, page-breaks, line-breaks, comments) may appear between any lexical token.

A.2 Reference Grammar

<x>	Means that x is a nonterminal
X	Means that x is a terminal
[x]	Means there are zero or one occurrences of x
x*	Means zero or more occurrences of x
x+,	Means a list of one or more comma separated occurrences of x
|	Separates alternatives
{}	Large braces are used for grouping

```
<program>          → <preprocessor>* <field_decl>* <method_decl>*
<preprocessor>     → #define <id> <literal_expression>
<field_decl>       → <type> (<var_decl_arg>)+, ;
<method_decl>      → {<type> | void} <id> ( [{<type> {<id> | <id>`[` `]` }}+,] )
                     <method_block>

<method_block>     → '{' <var_decl>* <statement>* '}'
<block>            → `{` <statement>* `}`
<if_block>         → `{` <if_statement>* `}`

<statement>        → <location> = <expr>; |
                     <location> <post_un_op>; |
                     <location> <arith_assign_op> <expr>; |
                     <method_call> ; |
                     if (<expr>) {<if_block> | <if_statement>}
                           [ else {<if_block> | <if_statement>} ] |
                     while (<expr>) { <block> | <statement> } |
                     for  ([<for_statement_1>+,];<expr>;[<for_statement_3>+,])
                           {<block> | <statement>}|
                     printf (); |
                     printf (<string_literal> [, <expr>+,] );  |

                     break ; |
                     return  [<expr>];

<if_statement>     → <location> = <expr>; |
                     <location> <post_un_op>; |
                     <location> <arith_assign_op> <expr>; |
                     <method_call> ; |
                     if (<expr>) {<if_block> | <if_statement>}
                           [ else {<if_block> | <if_statement>} ] |

                     while (<expr>) { <block> | <statement> } |

                     for  ([<for_statement_1>+,];<expr>;[<for_statement_3>+,])
                          {<block> | <statement>}|

                     printf (); |
                     printf (<string_literal> [, <expr>+,] );  |

                     return  [<expr>];


<for_statement_1>  → <location> = <expr> |
<for_statement_3>  → <location> = <expr> |
                     <location> <post_un_op> |
                     <location> <arith_assign_op> <expr> |
                     <method_call>

<location>         → <id> |
                     <id> `[` <expr> `]`
```

```
<expr>              → <location> |
                      <method_call> |
                      <literal_expression> |
                      <expr> + <expr> |
                      <expr> - <expr> |
                      <expr> * <expr> |
                      <expr> / <expr> |
                      <expr> % <expr> |
                      <expr> << <expr> |
                      <expr> >> <expr> |
                      <expr> & <expr> |
                      <expr> | <expr> |
                      <expr> ^ <expr> |
                      <expr> < <expr> |
                      <expr> > <expr> |
                      <expr> <= <expr> |
                      <expr> >= <expr> |
                      <expr> == <expr> |
                      <expr> != <expr> |
                      <expr> && <expr> |
                      <expr> || <expr> |
                      - <expr> |
                      ! <expr> |
                      ~ <expr> |
                      sin <expr> |
                      cos <expr> |
                      tan <expr> |
                      sqrt <expr> |
                      atan <expr> |
                      log10 <expr> |
                      log <expr> |
                      exp10 <expr> |
                      exp <expr> |
                      ( <expr> ) |
                      (<type) <expr>

<method_call>       → <id> (<expr>+,) | <id> ()

<var_decl>          → <type> {<var_decl_arg>}+, ;
<var_decl_arg>      → <id> |
                      <id> = <literal_expression> |
                      <id> `[` [<int_literal> | <char_literal>] `]` |
                      <id> `[` [<int_literal> | <char_literal>] `]` = '{' [{<number>}+,
                            | {<char_literal>}+,]  '}' |
                      <id> `[` [<int_literal> | <char_literal>] `]` = <string_literal>


<post_un_op>        → ++ | --
<arith_assign_op>   → += | -= | *= | /= | &= | |=  | ^= | >>= | <<=
<type>              → char | float | int | long
<id>                → <alpha> <alpha_num>*

<literal_expression>→ <number> | <char_literal> | <string_literal>
<number>            → <int_literal> | <longint_literal> | <hex_literal> |
                      <longhex_literal> | <bin_literal> | <longbin_literal> |
                      <float_literal>
<int_literal>       → <digit> <digit>*
<longint_literal>   → <digit> <digit>* [L|l]
<hex_literal>       → 0 x|X <hex_digit> <hex_digit>*
<longhex_literal>   → 0 x|X <hex_digit> <hex_digit>* [L|l]
<bin_literal>       → 0 b|B <bin_digit> <bin_digit>*
<longbin_literal>   → 0 b|B <bin_digit> <bin_digit>* [L|l]

<float_literal>     → <digit>* . <digit> <digit>* [e|E [-|+] <digit> <digit>*] |
                      <digit> <digit>* . <digit>* [e|E [-|+] <digit> <digit>*] |
                      <digit> <digit>* [e|E [-|+] <digit> <digit>*]
<char_literal>      → '<char>'
<string_literal>    → " <char>* "

<char>              → Any 8-bit ASCII character
```

```
<alpha_num>     → <alpha> | <digit>
<alpha>         → a | b | c | … | y | z | A | B | C | … | Y | Z | _
<digit>         → 0 | 1 | 2 | … | 9
<hex_digit>     → 0 | 1 | 2 | … | 9 | a | b | … | f | A | B | … | F
<bin_digit>     → 0 | 1
```

A.3 Precedence Definitions

Operator	*Associatively*
() []	*Left to right*
! ~ ++ -- -	*Right to left*
* / %	*Left to right*
+ -	*Left to right*
<< >>	*Left to right*
< <= > >=	*Left to right*
== !=	*Left to right*
&	*Left to right*
^	*Left to right*
|	*Left to right*
&&	*Left to right*
||	*Right to left*
= += -= *= /= &= |= ^= >>= <<=	*Right to left*
,	*Left to right*

A.4 Semantic Rules

1. Duplicate variable names within the same scope are not allowed.
2. A #define variable cannot share a name with any other variable regardless of the scope.
3. #define variables cannot be associated with non-char arrays.
4. No functions should share the same name.
5. All variables and functions need to be declared.
6. An array declaration must indicate its size. The declared size of the array can be an integer or a char literal.
7. Initial values for any variable declaration must be a literal value of the same type as the declared type of the variable.

8. The initial value list of an array must be less than or equal to the declared size of the array.
9. An array index must be an integer or a char literal.
10. Casting is only allowed to and from a float.
11. exp, exp10, sin, cos, tan, atan, log, log10, sqrt operations must be performed on floats
12. mod, bitnot, lognot, logor, logand, bitxor, bitor, bitand, >>, << operations can only be performed on int operands.
13. +, -, *, /, ==, !=, <, >, <=, >= operations can only be performed on operands of the same type
14. / operation cannot be used on a long
15. A return expression must be of the same type as the return type of the function.
16. If a function is declared to return something, then it must have a corresponding return statement in all possible execution paths.
17. The data types of the arguments in a printf statement must correspond to what is indicated in the string of the printf statement.
18. The number of arguments in a printf statement must correspond to what is indicated in the string of the printf statement.

Appendix B

ICB File Format

The ICB file format consists of three primary components:

- Binary code assembled in one location
- Binary code assembled in another location
- Internal references, variable references, and function references.

B.1 Binary Code

The binary code that is given in an ICB file is to be downloaded directly to the controller board. The code is found in two separate blocks that are built in two different addresses. This is done to allow the parser of an ICB file to be able to detect address locations within the ICB file. Any binary code that differs between the two blocks consists of code that refers to an address within the binary code. Such addresses need to be adjusted according to the final location of the binary code. The addresses are each two bytes long.

Each line of binary code follows a special format. The following lists what is found in a line of binary code:

- S1: The first item in a binary code line is the "S1" string. This string alerts whatever's parsing the ICB file that this line contains binary code

- Byte count: The byte count is a one-byte value of the number of bytes to be found after the last byte of the byte count.
- Address code was build: The address where this line of code was built is next. This is a two byte address value.
- Binary code: The remaining string of this line except for the last byte is the binary code.
- Checksum: The last byte of the line is the checksum. If everything on that line of code except for the "S1" string was added in one-byte blocks, the value should sum to something that ends in 0xFF in hexadecimal.

The first block of binary code consists of consecutive lines of binary code. The block of binary code is terminated by a line that starts with the string "S9" which indicates that that line is not a data line and should be ignores. That line after the "S9" line starts the second block of code that was built in a different address locations. This block is also terminated by a line that starts with "S9."

B.2. Internal references, variable references, and function references

Following the two blocks of code are various internal references. Most of these references can be ignored by the ICB parser since these references are not relevant to the final code that is loaded. However, found within these internal references are references to the variables and functions that can be referenced.

Variable references are start with the string "variable_" followed by the name of the variable. Similarly, function references start with string "subroutine_" followed by the name of the function. Following the name of the function or variable is the address

where the variable can be accessed or the function can be called. Everything else on that line is irrelevant and can be ignored.

This address is a two byte address that was built according to the address used to build the second binary code block. These addresses needed to be adjusted depending upon where the final binary code is loaded by the compiler. This information allows another program to be aware of what global variables and functions are available and the addresses to access them.

Appendix C

Available Pseudocode Instructions

C.1 General Instructions

Instruction	*Opcode Value*	*Arguments*	*Pops*	*Pushes*
add2	0	*None*	2 two byte operands	Result of + operation
sub2	2	*None*	2 two byte operands	Result of - operation
mult2	4	*None*	2 two byte operands	Result of * operation
div2	6	*None*	2 two byte operands	Result of / operation
bitand2	8	*None*	2 two byte operands	Result of & operation
bitor2	10	*None*	2 two byte operands	Result of | operation
bitxor2	12	*None*	2 two byte operands	Result of ^ operation
equal2	14	*None*	2 two byte operands	Result of == operation
lt2	16	*None*	2 two byte operands	Result of < operation
gt2	18	*None*	2 two byte operands	Result of > operation
lshift	20	*None*	2 two byte operands	Result of << operation
lognot2	22	*None*	1 two byte operand	Result of ! operation
logidn2	24	*None*	1 two byte operand	Result of logical identity operation
bitnot2	26	*None*	1 two byte operand	Result of ~ operation
neg2	28	*None*	1 two byte operand	Result of negation operation
add4	30	*None*	2 four byte operands	Result of + operation
sub4	32	*None*	2 four byte operands	Result of - operation
mult4	34	*None*	2 four byte operands	Result of * operation
lt4	36	*None*	2 four byte operands	Result of < operation
gt4	38	*None*	2 four byte operands	Result of > operation
equal4	40	*None*	2 four byte operands	Result of == operation
neg4	42	*None*	1 four byte operand	Result of negation operation
push2	44	1 two byte argument	*None*	Argument onto stack
push4	46	1 four byte argument	*None*	Argument onto stack
pushblock	48	No. of bytes to be pushed and inline data of bytes to be pushed	*None*	Bytes given in argument
pop2	50	*None*	Two bytes from stack	*None*
pop4	52	*None*	Four bytes from stack	*None*
peeki1	54	Two byte address	*None*	One byte value found at argument's address
peeki2	56	Two byte address	*None*	Two byte value found at argument's address

peeki4	58	Two byte address	*None*	Four byte value found at argument's address
peek1	60	*None*	Two byte address from stack	One byte value found at address found on stack
peek2	62	*None*	Two byte address from stack	Two byte value found at address found on stack
peek4	64	*None*	Two byte address from stack	Four byte value found at address found on stack
speek2	66	*None*	One byte offset	Two byte value that is the specified offset away from the stack pointer
speek4	68	*None*	One byte offset	Two byte value that is the specified offset away from the stack pointer
pokei1	70	Two byte address	One byte value to be written to address specified in argument	*None*
pokei2	72	Two byte address	Two byte value to be written to address specified in argument	*None*
pokei4	74	Two byte address	Four byte value to be written to address specified in argument	*None*
poke1	76	*None*	One byte value and two byte address, writes value to address specified	*None*
poke2	78	*None*	Two byte value and two byte address, writes value to address specified	*None*
poke4	80	*None*	Four byte value and two byte address, writes value to address specified	*None*
poke1nopop	82	*None*	One byte value and two byte address, writes value to address specified	One byte value that was popped
poke2nopop	84	*None*	Two byte value and two byte address, writes value to address specified	Two byte value that was popped
poke4nopop	86	*None*	Four byte value and two byte address, writes value to address specified	Four byte value that was popped
bitset	88	*None*	Two byte value and address, sets address location to value	*None*
bitclr	90	*None*	Two byte value and address, clears address location	*None*
spoke2	92	*None*	Two byte value and one byte offset	Two byte value just popped to specified offset from stack pointer
spoke4	94	*None*	Four byte value and one byte offset	Four byte value just popped to specified offset from stack pointer
jumpi	96	Two byte address to jump to	*None*	*None*
jump	98	*None*	Two byte address to jump to	*None*
jfalse	100	Two byte address to branch to	Two byte conditional, jumps if conditional is 0	*None*
jtrue	102	Two byte address to branch to	Two byte conditional, jumps if conditional is anything but 0	*None*
jpfalse	104	Two byte address to branch to	Two byte conditional, jumps if conditional is 0	If jump taken, pushes a 1, else pushes a 0
jptrue	106	Two byte address to branch to	Two byte conditional, jumps if conditional is anything but 0	If jump taken, pushes a 1, else pushes a 0
mret0	108	Number of bytes to pop off stack	Number of bytes indicated by argument. Then pops return address off stack and saves that as the program counter.	*None*
mret2	110	Number of bytes to pop off stack	Number of bytes indicated by argument. Then pops return address off stack and saves that as the program counter.	Pushes two byte return value on stack. The return value is saved before anything is popped.

mret4	112	Number of bytes to pop off stack	Number of bytes indicated by argument. Then pops return address off stack and saves that as the program counter.	Pushes four byte return value on stack. The return value is saved before anything is popped.
aref1	114	*None*	Two byte index and two byte address where array is located.	Address where a char array item indicated by index and array address is located.
aref2	116	*None*	Two byte index and two byte address where array is located.	Address where an int array item indicated by index and array address is located.
aref4	118	*None*	Two byte index and two byte address where array is located.	Address where a float or long array item indicated by index and array address is located.
addsp	122	A two byte argument. The current value of the stack is modified by the argument	*None*	*None*
sprel	124	A two byte argument	*None*	Sum of current value of stack pointer and argument
checkstack	126	A two byte argument. A run time error will be generated if the amount of free space available in the stack is less than what is indicated by the argument.	*None*	*None*
setsp	128	*None*	Two byte value, the stack pointer is set to that value.	*None*
fl2int	130	*None*	Pops a four byte float	Pushes two byte int representation of float value.
int2fl	132	*None*	Pops a two byte int.	Pushes flour byte float representation of int value.
fl2lng	134	*None*	Pops a four byte float.	Pushes four byte long representation of float value.
lng2fl	136	*None*	Pops a four byte long.	Pushes flour byte float representation of long value.
fadd	138	*None*	2 four byte operands	Result of + operation
fsub	140	*None*	2 four byte operands	Result of - operation
fmult	142	*None*	2 four byte operands	Result of * operation
fdiv	144	*None*	2 four byte operands	Result of / operation
fequal	146	*None*	2 four byte operands	Result of == operation
flt	148	*None*	2 four byte operands	Result of < operation
fgt	150	*None*	2 four byte operands	Result of > operation
fneg	152	*None*	1 four byte operand	Result of negation operation
fsqrt	154	*None*	1 four byte operand	Result of sqrt operation
fexp	156	*None*	1 four byte operand	Result of exp operation
f10tx	158	*None*	1 four byte operand	Result of exp10 operation
fx2y	160	*None*	2 four byte operands	Result of x to the power y
fln	162	*None*	1 four byte operand	Result of log operation
flog	164	*None*	1 four byte operand	Result of log10 operation
fatan	166	*None*	1 four byte operand	Result of atan operation
fsin	168	*None*	1 four byte operand	Result of sin operation
fcos	170	*None*	1 four byte operand	Result of cos operation
ftan	172	*None*	1 four byte operand	Result of tan operation

C.2 Special Instructions

Instruction	*Opcode Value*	*Description*
fl2ascii	174	Takes a floating point number and an address to a string, returns a string.
printlcd2	176	Pops two byte word and prints to LCD.
printf	178	Pops string and arguments and prints on LCD.
printstring	180	Pops address where string is located and prints that string on LCD.
printchar	182	Pops char off stack and prints on LCD.
startprocess	184	Special task control command.
haltnotify	186	Special task control command.
killprocess	188	Special task control command.
defer	190	Special task control command.
systime	192	Special task control command.
loadreg	194	Takes one byte argument specifying register to write to. Pops value at top of stack and writes to specified register.
fetchreg	196	Takes one byte argument specifying register to read from. Takes value at specified register and pushes onto stack..
bench	198	Returns integer indicating how many machine cycles were able to execute in one milliseconds of real time.
callml	200	Calls function specified in an ICB file. Pops two bytes representing argument to function and pops address to jump to. Jumps to address
initint	202	Initializes 6811 interrupts to pcode defaults, de-installing any binary module routines.
undefined	204	Undefined instruction.

Appendix D

Listing of Source Code

- ***Main Module***
  - ic.java: main file, contains "main" function to execute program
- ***Board Module***
  - board/board.java: contains all code for interfacing with controller board
- ***Compiler Module***
  - compiler/compiler.java: main file for compiler module
  - compiler/compilerInfo.java: data structure that holds information to be returned from compiler module to main module
  - compiler/compilerLISInfo.java: data structure that holds information to be returned from compiler module to main module when compiling a LIS file
  - compiler/errorMsg.java: data structure used to hold error messages
  - compiler/Opcode.java: file holding all translations from names of pseudocode commands to numeric representation of pseudocode command
  - compiler/OpType.java: file holding encodings of all operations (i.e. +, - *)
  - compiler/VarType.java: file holding encodings of all data types (i.e. int, long)
  - ***Scanner Sub-Module***
    - compiler/Ylex: JLex input file
  - ***Parser Sub-Module***
    - compiler/parse1.cup: stage one parser CUP input file
    - compiler/parse1Info.java: data structure holding information to be returned from stage one parser to compiler
    - compiler/parse2.cup: stage two parser CUP input file
    - compiler/parse2Info.java: data structure holding information to be returned from stage two parser to compiler
    - compiler/parseInteractiveInfo.java: data structure holding information to be returned from stage two parser to compiler after compiling interactively entered code
    - compiler/Expression.java: expressions used in creating parse tree
      - compiler/BinOpExpression.java
      - compiler/CastExpression.java
      - compiler/FunctionCallExpression.java
      - compiler/LiteralExpression.java
      - compiler/LocationExpression.java
      - compiler/PreUnOpExpression.java
    - compiler/Function.java: data structure used to describe a function
    - compiler/ProgramSignature.java: data structure representing a program signature

- compiler/Program.java: data structure representing a program, subclass of a program signature.
- compiler/Statement.java: statements used in creating parse tree
  - compiler/ArithAssignOpStatement.java
  - compiler/AssignmentStatement.java
  - compiler/BreakStatement.java
  - compiler/ForStatement.java
  - compiler/FunctionCallStatement.java
  - compiler/IfElseStatement.java
  - compiler/PostUnOpStatement.java
  - compiler/PrintfStatement.java
  - compiler/ReturnStatement.java
  - compiler/WhileStatement.java
- compiler/Variable.java: data structure used to describe a variable

- ***Flattener Sub-Module***
  - compiler/flatten.java: main file for flattener
  - compiler/flatProgram.java: data structure used to hold flattened program
  - compiler/Instruction.java: instructions used in flattened program
    - compiler/ArrayPointerInstruction.java
    - compiler/AssignArrayLocationInstruction.java
    - compiler/AssignArrayLocationInstruction1.java
    - compiler/AssignArrayLocationInstruction2.java
    - compiler/AssignBinOpInstruction.java
    - compiler/AssignFunctionCallInstruction.java
    - compiler/AssignLiteralInstruction.java
    - compiler/AssignLocationInstruction.java
    - compiler/AssignLocationInstruction1.java
    - compiler/AssignPreUnOpInstruction.java
    - compiler/AssignToArrayInstruction.java
    - compiler/BranchFalseInstruction.java
    - compiler/BranchPushFalseInstruction.java
    - compiler/BranchPushTrueInstruction.java
    - compiler/BranchTrueInstruction.java
    - compiler/CastInstruction.java
    - compiler/FunctionCallExpressionInstruction.java
    - compiler/FunctionCallInstructionBegin.java
    - compiler/FunctionCallStatementInstruction.java
    - compiler/FunctionCallEndInstruction.java
    - compiler/FunctionStartInstruction.java
    - compiler/JumpInstruction.java
    - compiler/LabelInstruction.java
    - compiler/LogidnInstruction.java
    - compiler/PrintfInstruction.java
    - compiler/ReturnInstruction.java
    - compiler/StringLiteralInstruction.java
    - compiler/VarDeclInstruction.java
      - compiler/GlobalVarDeclInstruction.java
      - compiler/LocalVarDeclInstruction.java

- ***Code Generator Sub-Module***
  - compiler/codeGen.java: main file for code generator
  - compiler/codeGenInfo.java: data structure that holds information that is returned from code generator to compiler module

References

1. Aho, Alfred; Sethi, Ravi; Ullman, Jeffrey. Compilers. Reading, Massachusetts: Addison Wesley. 1988.
2. Appel, Andrew. Modern Compiler Implementation in Java. Cambridge: Cambridge University Press. 1998.
3. Muchnik, Steven. Advanced Compiler Design & Implementation. San Francisco: Morgan Kaufmann. 1997.
4. http://www.cs.princeton.edu/~appel/modern/java/CUP/, *CUP Parser Generator for Java*
5. http://www.cs.princeton.edu/~appel/modern/java/JLex/, *JLex: A Lexical Analyzer Generator for Java*
6. http://www.handyboard.com, *The Handy Board*