Lab 3 Group BEERZ

Programmer's Guide

**Evan Todd**

**Elliot Schumacher, Brad Kline, Ryan Powers, and Zach Smith**

**3/11/2011**

**Introduction**

***~~This document contains the summary of the implementation of the “Wi-11 Machine” linking loader, and will cover all of the components that are used in the execution of the linking loader. The linking loader will accept one or more object files that have been assembled, properly combine the object files and produce a single object file that is capable of being run on the “Wi-11 Machine” simulator. Object files are to follow the format laid out in the Assembler User’s Guide; this document is written cased on the assumption that the reader is familiar with this format. Throughout the guide, names of classes and system defined types will be in Bold Courier Font. Classes and methods are public unless stated otherwise.~~***

**Linking Loader Overview**

***~~The linking loader is capable of handling multiple object files, and linking them together in order to produce a single executable file that can be input to the simulator. The linking loader is able to handle linking multiple relocatable programs, one relocatable program with no external symbols, and one absolute program with no external symbols. When the linking loader runs, the entry point is the class entitled Main, which is implemented in the Loader package in the Main.java file. Main is responsible for accepting and handling command line arguments and producing the simulator-executable object file.~~***

**Directory Structure**

**The loader's files are in a parent directory named “cse-560-beerz,” that contains two subfolders named Loader and Common (the Common subfolder contains classes that are used in all three components of the Wi-11 machine). The following files are in the Loader directory:**

1. **Main.java**
2. **Contains the implementation of the Main class, which handles all disk and console I/O and oversees the linking process.**
3. **Loader.java**
4. **Contains the implementation of the stateless Loader class, which parses object files into instances of the ObjectFile class.**
5. **ObjectFile.java**
6. **Implements the ObjectFile class, which represents an object file loaded in memory. This class stores an object file's segment name, execution start address, text records, imported and exported symbols, and relocation information.**

**The following files are in the Common directory:**

1. **ByteOperations.java**
2. **Implements the static ByteOperations utility class, which provides functions for modifying binary data.**
3. **Error.java**
4. **The Error class is used by the Assembler, Loader, and Simulator to keep track of errors encountered while assembling, loading, or simulating a program.**
5. **MemoryBank.java**
6. **A MemoryBank represents an array of 16-bit memory. It provides functions for reading and writing memory, as well as resolving symbol values and relocating binary data.**
7. **SymbolEntry.java**
8. **A SymbolEntry represents a location in memory that must be modified by the Loader. The SymbolEntry may be associated with a Symbol, in which case the Symbol's value must be inserted at the memory location. Otherwise, the SymbolEntry just indicates that the value in memory must be modified if it is relocated.**
9. **Symbol.java**
10. **A Symbol is a name/value pair with an extra flag that defines whether the Symbol is relocatable or constant.**
11. **SymbolTable.java**
12. **A SymbolTable is an internal representation mapping names to Symbols.**

**The Loader folder contains the subfolder “/tests” that contains the test assembly and object files that were used in the testing of the Loader.**

**The Loader folder also contains the subfolder “/docs” that contains the User’s Guide, Programmer’s Guide and the Testing Plan. Contained within the “/docs” directory is another subfolder “/Meeting Minutes” that contains the minutes for each design meeting that took place.**

**Design Conventions**

**Across the files in our implementation, several design conventions were followed. These include the capitalization of class names and package names, like Main or Loader, lower-case names for variables, and mixed-case names for method titles like “getSegmentName”. Also, care was taken to limit the length of lines to less than 90 characters. On a whole, our lab was designed to be object-oriented. The objects that are used to represent the symbols and object files follow the object-oriented convention of encapsulation and data hiding.**

**Module Inter-Relationships**

**The core linking loader algorithm is implemented in the Main and Loader classes. These modules use the ObjectFile, SymbolTable, Symbol, SymbolEntry, and MemoryBank classes to represent the state of the object files. In addition, the Error class is used for error reporting, and the ByteOperations utility class is used for binary modification functions.**

**Data Structures**

**There are several important data structures that are used by the loader in the creation of the output executable. The following is a listing of the data structure, the file it appears in, the usage of the data structure, its implementation, and its invariant.**

1. **Object: “ObjectFile” defined by ObjectFile.java**
2. **Represents a complete “Wi-11 Machine” object file, including the segment name, execution start address, text records, imported and exported symbols, and relocation information.**
3. **Implemented as a String for the segment name, an int for the start address, a MemoryBank for the text records, two Lists of SymbolEntrys for the imported and exported symbols and the relocation information.**
4. **Invariant: length of the segment name must be 6 and the start address must be in the range [x0, xffff]. The MemoryBank must not be null.**
5. **Object: “MemoryBank” defined by MemoryBank.java**
6. **Represents an array of 16-bit memory.**
7. **Implemented as a HashMap<Integer, Short> and two ints to track the first and last address containing data.**
8. **Invariant: the first and last addresses tracked by the MemoryBank must correspond to the smallest and largest int keys (respectively) in the HashMap<Integer, Short>.**
9. **Object: “SymbolEntry” defined by SymbolEntry.java**
10. **Represents a location in memory that must be modified by the Loader.**
11. **Implemented as a String that may be null for relocation records, or may contain the name of a Symbol associated with the entry, and three ints which represent a memory address and the indices of the most significant and least significant bits that need modified.**
12. **Invariant: the address must be in the range [x0, xffff], the index of the most significant bit must be larger than that of the least significant bit. Both bit indices must be in the range [0, 16].**
13. **Object: “Error” defined by Error.java**
14. **Represents a loader error with a message and an optional object file line number.**
15. **Implemented as a String to store the message, an int to store the line number, and a boolean to indicate whether the error makes use of the line number.**
16. **Invariant: the message must not be null, and the line number must be greater than zero.**
17. **Object: “Symbol” defined by Symbol.java**
18. **Represents the name/value pair of a defined symbol.**
19. **Implemented as a String for the symbol name, an int to store its value, and three booleans indicating whether the symbol is relocatable, imported, or exported.**
20. **Object: “SymbolTable” defined by SymbolTable.java**
21. **Represents the internal mappings of names to Symbols.**
22. **Implemented as a HashMap<String, Symbol>.**
23. **Invariant: Only unique Symbols can be entered into the SymbolTable.**

**Component Descriptions**

**This section provides a detailed description of both the client- and implementation-side view of every component used in the program.**

**Client-Side**

1. **Main**
2. **Description: The Main component is the entry point into the loader program. It accepts the command line arguments from the user, such as the input files, output filename, and an option to generate a listing.**
3. **Loader**
4. **Description: The Loader component parses object files into in-memory ObjectFile reperesentations, which can then be relocated and linked.**
5. **Mathematical Model:**

**The Loader has no state.**

1. **Constraint:**
2. **ObjectFile**
3. **Description: The ObjectFile component holds the in-memory representation of an object file. The ObjectFile can then be relocated and linked into an executable file.**
4. **Mathematical Model:**

**ObjectFile = Start Address + Segment Name + MemoryBank + Symbol Entries + Relocation Records + Relocatable**

**Implementer-Side**

**Main**

1. **Description: The Main class contains the main method which is the entry point into the assembler program.**
2. **State: None**
3. **Algorithm:**

**If ( |args| < 2 OR args[0] = “--help” OR args[0]=”/h”) then**

**Display usage information**

**For int x = 2, x < |args| do**

**If ( args[x] = “-l” then**

**x++ and look at next args[x]**

**Set generateListing to true**

**Else**

**Display usage information**

**filename = args[0]**

**outfile = args[1]**

**load all of filename into a string**

**create new Assembler**

**assemble code**

**generate code**

**output code to outfile**

**close output streams**

1. **Main Methods**
2. **Method Name: Main**
3. **Description: main class in class Main**
4. **Parameteres: args[]**
5. **Requires: input file**
6. **Alters: outfile**
7. **Returns: void**
8. **Throws: IOException**
9. **Method Name: printUsageInformation**
10. **Description: Prints usage information to the console. Shows:**

**"Usage: java Main inputfile outputfile [options]**

**Inputfile Specify path to input assembly file.**

**Outputfile Specify path to output object file.**

**-l Generate and display source code listing."**

1. **Parameters: None**
2. **Alters: System.out**
3. **Ensures: System.out = #System.out + [usage info]**
4. **Returns: void**
5. **Method Name: readAllText**
6. **Description: Reads all text in the file at the given path location into a string.**
7. **Parameters: String filename - Path to desired file**
8. **Requires: valid file**
9. **Returns: A string containing all the data existing in the desired file.**
10. **Throws: IOException**
11. **Method Name: writeAllText**
12. **Description: Writes all the given text to the file existing at the path location**
13. **Parameters: String filename - Desired output file name, String data – data to be written to file**
14. **Alters: The specified file on the file system**
15. **Ensures: The specified file is overwritten or created if it does not exist on the file system**
16. **Returns: void**
17. **Throws: IOException**

**Assembler**

1. **Description: The Assembler uses a constant instruction definition table to complete the first pass of the assembly process.**
2. **State: none**
3. **Alogrithm:**

**begin**

**foreach (line in lines)**

**if (first char is ‘;’)**

**line is a comment**

**increment line number**

**continue**

**endif**

**if (chars between label, operation, and operands != spaces)**

**add new Error message at current line number**

**if (Symbol Table contains this symbol)**

**symbol already exists**

**add new Error message at current line number**

**endif**

**else**

**if (instruction != “.EQU”)**

**add symbol to table**

**endif**

**if (instruction is “.ORIG”)**

**if (“.ORIG” is properly formed)**

**store segment information**

**else**

**add Error message**

**endif**

**else**

**if (instruction = “.EQU”)**

**if (“.EQU” is properly formed)**

**add to Symbol Table**

**else**

**add Error message**

**endif**

**else**

**if (instruction = “.FILL”)**

**if (“.FILL” is properly formed)**

**create new Operand**

**else**

**add Error message**

**endif**

**else**

**if (instruction = “.STRZ”)**

**if (“.STRZ” is properly formed)**

**create series of Operands representing the string ending with last Operand equal to 0x000**

**else**

**add Error message**

**endif**

**else**

**if (instruction = “.END”)**

**if (“.END” is properly formed)**

**add to the Symbol Table**

**else**

**add Error message**

**endif**

**else**

**if (instruction = “.BLKW”)**

**if (“.BLKW” is properly formed)**

**create Operand and update Location Counter to next position**

**after .BLKW**

**else**

**add Error message**

**endif**

**else**

**invalid instruction**

**add Error message**

**endif**

**if (program length exceeds one page of memory)**

**add Error message**

**endif**

**if (program loads outside of addressable memory range)**

**add Error message**

**endif**

**if (Error messages > 0 )**

**output Error messages**

**endif**

**return new program**

**end**

1. **Assembler Methods**
2. **Method Name: assemble**
3. **Description: This method is the first pass of the assembler that begins the translation of the source code into the object file**
4. **Parameters: String filename, String data**
5. **Requires: A valid filename**
6. **Ensures: data is a properly formed “Wi-11” assembly program**
7. **Returns: Program**
8. **Throws: Exception**
9. **Method Name: getInstructionDefinition**
10. **Description: This method finds an InstructionDefinition in the definition table for the given Instruction**
11. **Parameters: Instruction instruction**
12. **Ensures: InstructionDefinition is an acceptable definition**
13. **Returns: InstructionDefinition definition**
14. **Method Name: getOperands**
15. **Description: This Method extracts the raw string values of the Operands in a given line of source code.**
16. **Parameters: String line**
17. **Requires: A valid String**
18. **Ensures: line is a valid input**
19. **Returns: String[]**
20. **Throws: Exception**
21. **Method Name: getFileFromFileName**
22. **Description: This method returns a usable segment name from the given filename**
23. **Parameters: String filename**
24. **Returns: String – a 7 character string**

**ByteOperations**

1. **Description: A utility class that is used to obtain specific bits in a given hexadecimal number.**
2. **ByteOperations Methods:**
3. **Method: parseHex**
4. **Description: This method converts a hex number represented in a String to an integer value.**
5. **Parameters: String hex**
6. **Ensures: hex is a valid hexadecimal number representation.**
7. **Returns: an integer value representing the input hex value.**
8. **Method: extractValue**
9. **Description: This method extracts bit values from a given integer.**
10. **Parameters: int value, int start, int end**
11. **Requires: start < end**
12. **Returns: an integer value representing the extracted bits.**
13. **Method: getHex**
14. **Description: This method gets a string representation of a given integer value.**
15. **Parameters: int value, int numCharacters**
16. **Ensures: All but the least significant hex digits specified by numCharacters are removed.**
17. **Returns: a string representation of the integer value in hexadecimal form.**
18. **Method: getBit**
19. **Description: This method returns true if the bit in “value” at the specified right-aligned index is 1.**
20. **Parameters: int value, int index**
21. **Returns: A boolean value indicating if the specified bit is 1.**

**Instruction**

1. **Description: Each instance of this class represents an assembly instruction that contains a definition, a name and a collection of Operands.**
2. **State:**
3. **private InstructionDefinition definition**
4. **private Operand[] operands**
5. **private String source**
6. **private String name**
7. **Instruction Methods**
8. **Method: setDefinition**
9. **Description: Sets the definition of this Instruction, as well as the definition of all this Instruction's Operands. Note: setOperands must be called first.**
10. **Parameters: InstructionDefinition definition**
11. **Requires: A valid OperandDifinition[]**
12. **Returns: void**
13. **Throws: Exception**
14. **Method: setOperands**
15. **Description: This method initializes the values of the Instruction’s Operands using the given values extracted from the source code**
16. **Parameters: String[] values, LiteralTable literals**
17. **Returns: void**
18. **Throws: Exception**
19. **Method: getSource**
20. **Description: This method returns the line of assembly source code associated with this Instruction**
21. **Returns: String**
22. **Method Name: getName**
23. **Description: This method returns the name of the Instruction**
24. **Returns: String**
25. **Method Name: getDefinition**
26. **Description: This method returns the definition of the Instruction**
27. **Returns: InsructionDefiniton**
28. **Method Name: getOperands**
29. **Description: This method returns the collection of Operands for the Instruction**
30. **Returns: Operands[]**
31. **Method Name: getCodes**
32. **Description: This Method gets the final executable binary codes called for by the Instruction**
33. **Parameters: SymbolTable symbols, LiteralTable literals**
34. **Returns: int[] result**
35. **Throws: Exception**

**Error**

1. **Description: An instance of Error describes an error encountered in the assembly of the input program.**
2. **State: private int line, private String message, private boolean hasLine**
3. **Error Methods:**
4. **Method Name: hasLineNumber**
5. **Description: Used to determine whether this is associated with a line number.**
6. **Returns: true if and only if this error has a line number associated with it**
7. **Method Name: getLineNumber**
8. **Description: Gets the line number associated with this Error.**
9. **Returns: 0 if this.hasLine = false; otherwise returns the line number of this Error**
10. **Method Name: getMessage**
11. **Description: Gets the message describing this Error to the user.**
12. **Returns: the message associated with this Error**

**InstructionDefinition**

1. **Description: An InstructionDefinition represents a certain type of assembly instruction (like ADD, LD, STR, etc.). Once an assembly instruction is matched with a definition, the definition is used to turn the source code into binary executable code. An InstructionDefinition is comprised of: A name, used to match instruction names extracted from source code lines. A collection of binary values which represent the basis of the final binary code; this is the binary executable code before the Operand values are inserted. A collection of OperandDefinitions which specify which Operands are acceptable for this instruction, and how they are to be inserted into the binary base values.**
2. **State:**
3. **protected static InstructionDefinition[] definitions**
4. **Private int[] operations**
5. **A collection of binary values which represent the basis of the final binary code.**
6. **private OperandDefintion[] operands**
7. **A collection of OperandDefinitions which specify which Operands are acceptable**

**for this instruction, and how they are to be inserted into the binary base values.**

1. **private String name**
2. **The name of this instruction type, used to match instruction names extracted from source code lines.**
3. **private int size**
4. **Represents the number of memory slots this instruction takes up. May not correspond to the size of the operations array.**
5. **InstructionDefiniton Methods**
6. **Method Name: getTable**
7. **Description: This method returns and array of instruction definitions representing the instruction table**
8. **Returns: InstructionDefinition[]**
9. **Method Name: isAcceptable**
10. **Description: This method returns true if the given Instruction (presumably initialized with a name and a collection of Operands) is acceptable for this definition. Namely, it must have the same name as this definition, it must have exactly one Operand for each OperandDefinition in this definition, and the Operands must be acceptable for all the OperandDefinitions.**
11. **Parameters: Instruction instructions**
12. **Returns: true if and only if the instruction is a valid instruction name.**
13. **Method Name: getOperations**
14. **Description: This is the binary executable code before the Operand values are inserted.**
15. **Returns: int[]**
16. **Method Name: getOperandDefinitions**
17. **Description: This method gets the collection of OperandDefinitions for this definition**
18. **Returns: OperationDefinition[]**
19. **Method Name: getRelocationMasks**
20. **Description: This method masks An array of bitmasks which specify which parts of the instruction**
21. **Returns: int[] masks**
22. **Method Name: getName**
23. **Description: This method gets the name of this definition. Used to match instruction names extracted from source code lines.**
24. **Returns: String**
25. **Method Name: getSize**
26. **Description: This method returns an integer representing the number of address slots this instruction requires.**
27. **Returns: int**

**LiteralTable**

1. **Description: Represents a table of literals accumulated during the first assembler pass.**
2. **State:**
3. **protected static final int MIN\_VALUE**
4. **protected static final int MAX\_VALUE**
5. **private int index**
6. **private Map<Integer, Integer> map**
7. **private int offset**
8. **LiteralTable Methods:**
9. **Method Name: setOffset**
10. **Description: Sets the address (relative to the program origin) of the first literal in this table.**
11. **Parameters: int offset**
12. **Alters: this**
13. **Ensures: this.offset = offset**
14. **Method Name: getOffset**
15. **Description: Gets the address (relative to the program origin) of the first literal in this table.**
16. **Returns: int this.offset**
17. **Method Name: define**
18. **Description: Defines a new literal with the given integer value.**
19. **Parameters: int value**
20. **Alters: this**
21. **Ensures: this.map = {#this.map} + {value}**
22. **Throws: Exception e**
23. **Method Name: getAddress**
24. **Description: Gets the address (relative to the program origin) where the given literal value is stored.**
25. **Parameters: int value**
26. **Returns: int address of value**
27. **Method Name: getEntries**
28. **Description: Gets the set of all address/value pairs in this table.**
29. **Returns: Set<Map.Entry<Integer, Integer>>**

**Operand**

1. **Description: An Operand instance represents a value (a Symbol, register identifier, literal, or immediate value) to be inserted into an instruction.**
2. **State: private OperandDefinition definition, private String value, private OperandType type**
3. **Operand Methods:**
4. **Method: setDefinition**
5. **Description: Sets the definition of this Operand. No work is done at this point, except some basic error checking for immediate values.**
6. **Parameters: OperandDefinition definition**
7. **Ensures: this.definition = definition**
8. **Alters: this**
9. **Throws: Exception e**
10. **Method: getType**
11. **Description: Returns what type of Operand this is.**
12. **Returns: OperandType the type of the current Operand**
13. **Method: insert**
14. **Description: Inserts the binary value of this Operand into the given instruction (ops). The given SymbolTable and LiteralTable are used to resolve the binary value of the Operand.**
15. **Parameters:**
16. **int[] ops**
17. **the op code of the assembly instruction**
18. **SymbolTable symbols**
19. **the SymbolTable used to create this instruction**
20. **LiteralTable literals**
21. **the LiteralTable used to create this instruction**
22. **Throws: Exception e**
23. **Method: determineType**
24. **Description: Given the string representation of an Operand, returns the OperandType that describes it.**
25. **Parameters: String value**
26. **Returns OperandType**
27. **Method: getValue**
28. **Description: Gets the binary Operand value represented by the given string. The SymbolTable and LiteralTable are used to resolve the binary value. Note: relocatable immediate values are NOT relocated by this function. No OperandDefinition is taken in this version, so no sanity checking is done.**
29. **Parameters: String value, SymbolTable symbols, LiteralTable literals**
30. **Returns: int the value of the Operand**
31. **Throws: Exception e**
32. **Method: getValue**
33. **Description: Gets the binary Operand value represented by the given string. The SymbolTable and LiteralTable are used to resolve the binary value. Note: relocatable immediate values are NOT relocated by this function.**
34. **Parameters: String value, SymbolTable symbols, OperandDefinition definition, LiteralTable literals**
35. **Returns: int the binary value of the Operand**
36. **Throws: Exception e**
37. **Method: parseConstant**
38. **Description: Parses a register identifier, literal value, decimal value, or hexadecimal value into an integer number.**
39. **Parameters: String value**
40. **Returns: int value of the Operand**
41. **Throws: Exception e**

**OperandDefinition**

1. **Description: An OperandDefinition states which values are acceptable for Operands that use this definition, and where to insert the binary value into the resulting executable code.**
2. **State:**
3. **private int mostSignificant**
4. **Right-aligned index of the most significant bit this definition's Operands are allowed to affect in the executable code.**
5. **private int leastSignificant**
6. **Right-aligned index of the least significant bit this definition's Operands are allowed to affect in the executable code.**
7. **private int opIndex**
8. **For Instructions that map to multiple memory addresses, this index identifies which memory address to insert the Operand into.**
9. **private boolean relocatable**
10. **True if this definition's Operands are relocatable, false if not.**
11. **private OperandType[] acceptableTypes**
12. **Any Operands that use this definition must have a type that is contained in this collection.**
13. **OperandDefinition Methods:**
14. **Method: isAcceptable**
15. **Description: Returns true if the given Operand meets the requirements of this definition. Namely, the type of the Operand must be contained in this definition's collection of acceptable OperandTypes.**
16. **Parameters: Operand operand**
17. **Returns: boolean true if and only if the Operand is valid**
18. **Method: isAcceptable**
19. **Description: Returns true if the given OperandType is contained in this definition's collection of acceptable OperandTypes.**
20. **Parameters: OperandType type**
21. **Returns: boolean true if and only if the OperandType is valid**
22. **Method: getMask**
23. **Description: Returns a bitmask which, when logically anded together with a binary Operand value, will zero out any bits which are not allowed to be affected by the value.**
24. **Returns: int bitmask to eliminate unused bits in the Operand**
25. **Method: getMostSignificantBit**
26. **Description: Gets the right-aligned index of the most significant bit this definition's Operands are allowed to affect in the executable code.**
27. **Returns: int the Operand’s most significant bit**
28. **Method: getLeastSignificantBit**
29. **Description: Gets the right-aligned index of the least significant bit this definition's Operands are allowed to affect in the executable code.**
30. **Returns: int the Operand’s least significant bit**
31. **Method: getMinimumAllowedValue**
32. **Description: Gets the minimum value this definition's Operands are allowed to have.**
33. **Returns: int the minimum allowed Operand value**
34. **Method: getMaximumAllowedValue**
35. **Description: Gets the maximum value this definition's Operands are allowed to have.**
36. **Returns: int the maximum allowed Operand value**
37. **Method: isSigned**
38. **Description: Returns true if and only if this definition accepts signed Operand values.**
39. **Returns: boolean indication of whether this definition accepts signed Operand values**
40. **Method: getOperationIndex**
41. **Description: For Instructions that map to multiple memory addresses, this index identifies into which memory address to insert the Operand.**
42. **Returns: int the memory address into which to insert the Operand**
43. **Method: isRelocatable**
44. **Description: Returns true if this definition’s Operands are relocatable, false if not.**
45. **Returns: boolean true if and only if the Operand is relocatable**
46. **Method: getAcceptableTypes**
47. **Description: Any Operands that use this definition must have a type that is contained in this collection.**
48. **Returns: OperandType[] of valid OperandTypes**

**OperandType**

1. **Description: Enumerated type that establishes the possible operand types:**
2. **IMMEDIATE**
3. **represents a constant value**
4. **REGISTER**
5. **corresponds to a register R0-R7**
6. **LITERAL**
7. **references a literal**
8. **SYMBOL**
9. **references a Symbol defined in the SymbolTable**
10. **STRING**
11. **used for the .STRZ pseudo-operation**

**Program**

1. **Description: A Program instance contains an in-memory representation of an assembly program, which can be rendered into binary form with the getCode function.**
2. **State:**
3. **private SymbolTable symbols**
4. **Contains the names and values of all Symbols defined in this Program.**
5. **private LiteralTable literals**
6. **Contains the addresses and values of all literals defined in this Program.**
7. **private List<Instructions> instructions**
8. **Contains all Instructions (source code, binary code, and Operand values) in this Program.**
9. **private int startAddress**
10. **The address (relative to the origin) at which to start execution.**
11. **private int origin**
12. **The absolute address of the first memory slot used by the Program. Any relocatable values in the Program are offset by this address.**
13. **private String name**
14. **Name for the program. Used as the segment name in the header record.**
15. **private boolean isRelocatable.**
16. **True if the program is relocatable.**
17. **Program Methods:**
18. **Method: getCode**
19. **Description: Gets the object code for this, optionally displaying a listing for the user.**
20. **Parameters: boolean printListing**
21. **Returns: a String representing the object file; if (printListing) then outputs the listing to the screen.**
22. **Throws: Exception e**

**Symbol**

1. **Description: A Symbol is a name-value pairing that has a flag for whether or not the symbol is relocatable or absolute.**
2. **State: private String name, private int value, private boolean isRelocatable**
3. **Symbol Methods:**
4. **Method: getName**
5. **Description: Returns the name of this Symbol**
6. **Returns: String this.name**
7. **Method: getValue**
8. **Description: Gets the value of this Symbol**
9. **Returns: int this.value**
10. **Method: isRelocatable**
11. **Description: Returns true if this Symbol is relocatable, false if not.**
12. **Returns: boolean this.isRelocatable**

**SymbolTable**

1. **Description: An instance of this class maintains a mapping between Symbols and their values.**
2. **State: private Map<String, Symbol>**
3. **SymbolTable Methods:**
4. **Method: define**
5. **Description: Adds a given Symbol to this SymbolTable.**
6. **Parameters: Symbol symbol**
7. **Alters: this**
8. **Ensures: this = #this + symbol**
9. **Method: define**
10. **Description: Defines an alias for another Symbol.**
11. **Parameters: String alias, String target**
12. **Alters: this**
13. **Ensures: this = #this + alias**
14. **Method: hasSymbol**
15. **Description: Returns true if this table contains a Symbol that matches the given name.**
16. **Parameters: String name**
17. **Returns: boolean true if and only if name matches a Symbol in this**
18. **Method: get**
19. **Description: Gets the Symbol mapped to the given name, or null if none exists in this.**
20. **Parameters: String name**
21. **Returns: Symbol corresponding to name input**
22. **Method: size**
23. **Description: Gets the number of defined Symbols in this table.**
24. **Returns: int number of defined Symbols**