CS 6353 Compiler Construction – Project Assignments

The goal of the projects discussed in this handout is to reinforce the compiler theory you learnt in class and familiarize you with some important tools that can help you implement a compiler. The programming languages you need to use to implement the compiler include C and C++ as well as the languages defined by lex and yacc.

Here are some general rules about the projects.

- students are not allowed to discuss among themselves in any way about project assignments. Questions regarding assignments can be discussed with the TA or the instructor. At the end of the semester, we will conduct plagiarism check on all projects. Students who are involved in plagiarism will get an F grade for the course. The penalty will be the same no matter who did the original work.
- Debugging is a part of the learning process and should be done by the student independently. Please do not seek help from TA or instructor or other students for debugging.
- You are responsible for generating your own testing programs and test your project thoroughly. We will use a different set of input data for testing, which will not be provided in advance.
- For each required output, it is your responsibility to print it in an easy to read format so that TA can see whether your code works or not. You have to eliminate debugging output. The output quality will be considered in grading.
- If your program does not fully function, please try to have partial results printed clearly to obtain partial credits. If your program does not work or it does not provide any correct output, then there will be no partial credits. If your program does work fully or partially but we cannot make it run properly, then it is your responsibility to make an appointment with the TA and come to the department to demonstrate your program. Otherwise, there will be no partial credits for the parts that do not work.
- Some requirements in the project specifications are set for specific learning objectives, while others are for ease of grading. Deviating from the requirements and specifications, irrespective of their objectives, will result in point deductions.
- You should submit your project files through elearning. The files that should be included are discussed in each project section. The standard platform for this project is the department servers csgrads1 and csgrads2. We will test your code on one of these two servers. You should make sure that your program runs on either of them.

In Section 1, we define two new languages by specifying their syntax and tokens. One of the languages is a machine code, MC. The other is a high level language FP, which has the functional languages’ syntax, but has the procedural languages’ characteristics. We also introduce a simplified version of FP, the SFP. Our goal is to construct a compiler which translates the FP language to the MC language using tools lex and yacc. But to make you understand how to develop lex and yacc, you will also try to achieve yacc like capability by your own coding. Section 2 onwards specify the projects.
1 Language Definitions

1.1 Definitions for FP

<program> ::= CONSTANTS <constant-definitions>
   FUNCTIONS <function-definitions>
   MAIN <statements>

<constant-definitions> ::= <constant-definition>*
<constant-definition> ::= { <constant-name> <scalar-parameter> }
<constant-name> ::= <identifier>
<scalar-parameter> ::= <number> | <Boolean> | <string>

<function-definitions> ::= <function-definition>*
<function-definition> ::= { <function-name> <arguments> return <return-arg> <statements> }
<arguments> ::= <argument>+<argument>
<argument> ::= <identifier>
<return-arg> ::= <identifier>

<statements> ::= <statement>*
<statement> ::= <assignment-stmt> | <read-stmt> | <print-stmt>
   | <if-stmt> | <loop-stmt> | <while-stmt>

<assignment-stmt> ::= { = <identifier> <parameter> }
<read-stmt> ::= { read <identifier>+ }
<print-stmt> ::= { print <print-parameter>+ }
<parameter> ::= <function-call> | <identifier> | <number>
<function-call> ::= { <function-name> <parameters> }
<function-name> ::= <identifier> | <predefined-function>
<predefined-function> ::= + | * | - | / | %
<parameters> ::= <parameter>*
<print-parameter> ::= <identifier> | <scalar-parameter>

<if-stmt> ::= { if <expression> then <statements> else <statements> }
<loop-stmt> ::= { loop <identifier> <statements> }
<while-stmt> ::= { while <expression> <statements> }
<expression> ::= { <comparison-operator> <parameter> <parameter> }
   | { Boolean }
<comparison-operator> ::= == | > | < | >= | <= | !=

Note that the angle brackets <>, parentheses (), ::=, and | are part of the BNF, but the curly brackets {} are part of the FP language. + and * are used in both BNF and FP, depending on the context.

1.2 Definitions for the Remaining Tokens in FP

An identifier has the form [a-z]+.
A number can be an integer or a float. An integer can have no sign or a negative sign (–) but no positive sign. It can be with or without space(s) between the sign and the digits. If the value is 0, only a single digit 0 is allowed. Otherwise, it is the same as common integer definitions (no leading 0’s). To avoid the overflow problem, the absolute value of an integer should be bounded. We assume that the input integer number given in the FP program should be bounded by 10,000,000 (it is not the bound during execution). A float has to have the decimal point and at least one digit on each side. The left side of the decimal point follows the rule for integers without the bound. The right side of the decimal point can be any number of digits but should have at least one digit.

A character string is enclosed within (). It can be [a-z, A-Z, 0-9, \n]+. For example, (I like lex and yacc) is a character string. Note that “space” is allowed in the string. We use \n to represent a new line. The character strings are only used in the print function.

A Boolean can only be “T” or “F”, where T represents true and F represents false.

Please also note that the FP language is case sensitive. Moreover, to ensure the correctness in processing the input FP-program, you need to consider white space. Treat the white space [ \t\n] the same way as in other high level languages such as C++ and Java, skip them.

### 1.3 Some Semantics of FP

Each of the predefined comparison operations takes two parameters and the corresponding functionality follows the conventional definition. Both parameters are float (or subtype integer). Similar to functional languages, a function call statement (to a user defined or predefined function) can be represented by its return value. For example, {+ A B} returns A+B, and it can also be viewed as a value that can be used as a parameter in other function calls. We only give a general function call definition, but each function can have its own constraints on its parameters, which is only specified in the semantic of the function.

The predefined functions + and * can take two or more parameters and return the sum and product of them, respectively. For example, {+ A B C} returns A + B + C. The predefined functions –, /, and % can only take two parameters. The parameters for % have to be integer and the returned value is also an integer. The parameters and the return value for +, –, *, / can be integer or float. If both parameters of / are integers, the returned value is an integer (with decimal points truncated); otherwise, the returned value is a float. Predefined functions have a similar syntactical structure as the user-defined functions. But they should not be treated as functions. There should be no call stack for them.

The statement “print” takes one or more parameters. It prints the value of each parameter. The statement “read” takes one or more identifiers. It simply reads a value from the standard input for each identifier. We assume that the input can only be a number, but print parameter can also be a string or a Boolean.

Note that even though from the syntax specification, a function call can have character string as a parameter (through constant definitions), we only allow print function to take string(s) as a
parameter. The other functions, including pre-defined and programmer defined functions, cannot have string as a parameter. A string cannot be assigned to an identifier either (but is allowed in constant definition, which is not an assignment).

The loop statement takes an <identifier> to indicate the number of iterations for executing the following <statements>. The identifier can only be an integer.

For function definitions, the final value of the <return-arg> should be returned automatically at the end of the function. There will be no specific return statement in the function body. Also, all the arguments are passed by value only.

1.4 Definitions of SFP

We modify FP into SFP. The definitions of SFP are as follows.

<program> ::= <statements>
<statements> ::= <statement>+ 
<statement> ::= { = <identifier> <parameter> } | 
{ loop <identifier> <statements> } 
{ print <parameter> }
<parameter> ::= <function-call> | <identifier> | <number>
<function-call> ::= { <function-name> <parameter> <parameter> } 
<function-name> ::= + | * 

The identifier and number definitions for SFP are the same as those in FP. Note that the negative sign “–” in SFP is important. There are no character strings or Boolean in SFP.

1.5 Definitions of MC

<program> ::= <instruction> ; <program> | <instruction> ; 
<instruction> ::= 
  <load-instruction> | <store-instruction> | 
  <add-instruction> | <sub-instruction> | 
  <mul-instruction> | <div-instruction> | <trunc-instruction> | 
  <if-instruction> | <test-instruction> | <goto-instruction> | 
  <read-instruction> | <print-instruction> 

<load-instruction> ::= load <register> <load-parameter> 
<store-instruction> ::= store <M-addr> <store-parameter> 
<add-instruction> ::= add <register> <register> <parameter> 
<sub-instruction> ::= sub <register> <register> <parameter> 
<mul-instruction> ::= mul <register> <register> <parameter> 
<div-instruction> ::= div <register> <register> <parameter> 
<trunc-instruction> ::= trunc <register> 
<goto-instruction> ::= goto <instrNum> 
<if-instruction> ::= if <register> <instrNum> 
<test-instruction> ::= <test-operator> <register> 
<read-instruction> ::= read <M-addr>
1.6 Token Definitions for MC

Note that some tokens are already used plainly in the syntax definition of the language. The tokens that are not defined in the syntax definition are defined as follows.

<register> should begin with character R, followed by a number, which can only be 0-9 (i.e., there are only 10 registers, R0, ..., R9).

<M-addr> should be expressed as M[x], where x can be a positive integer or a register with its content being a positive integer. If you have 100 variables, <M-addr> should be M[0] to M[99] (and they have to be consecutive).

<number> in MC is the same as number in FP.

<string> in MC is the same as string in FP.

<instrNum> should be a positive integer (starting from 1), pointing to an instruction in the MC program. It can be specified by a positive integer or a register. If it is specified by a register, then the content of the register has to be a positive integer.

<test-operator> in MC is the same as <comparison-operator> in FP.

1.7 Some Semantics for MC

<load-instruction> loads a memory content <M-addr> or a number to a register <register>.

<store-instruction> stores a register content <register> or a number to a memory slot <M-addr>.

<add-instruction> computes <register> + <parameter> and stores the result to the first <register>.

<sub-instruction> computes <register> – <parameter> and stores the result to the first <register>.

<mul-instruction> computes <register> * <parameter> and stores the result to the first <register>.

<div-instruction> computes <register> / <parameter> and stores the result to the first <register>.

<trunc-instruction> truncates the content of <register> and stores the result back to <register>.

<goto-instruction> simply causes the program to jump to the <instrNum>-th instruction.

<test-instruction> tests the relations == | > | < | >= | <= | != between the second <register> and the <parameter>. If true then the first <register> is set to 1; otherwise, the first <register> 0.

<if-instruction> tests if <register> content is positive (true). If so, then the program jumps to the <instrNum>-th instruction. If not, then it continues to the next instruction.

<read-instruction> reads from the standard input into the memory addresses M-addr. We assume that input data can only be <number>.

<print-instruction> prints the list of parameters. It can take number (integer or float), string, and <M-addr> as the parameters and print them.

<M-addr> is the address of the referenced identifier. You need to calculate and assign this address properly during code generation.
2 Project Lex -- Lexical Analysis of FP using lex

There are compiler-compiler tools that can make compiler construction much easier. We are going to use lex for token recognition. You are to define the tokens in the FP language in lex definitions and feed it to lex to generate a scanner (lexer). Then, you can use the generated scanner to process input FP programs. You need to insert appropriate code (actions) in your lex definition file so that the lexer (created by lex) will generate and print the tokens (token type and the original token string) appropriately.

Your program should print out the token type and the token string for each token. Token types include integer, float, string, Boolean, identifier, keyword (CONSTANTS, FUNCTIONS, MAIN, loop, if, while, =, read, print, return, etc.), predefined-function (+, *, -, /, %, etc.), comparison-operator (==, >, <, >=, <=, !=), and {, } (special characters).

If your scanner encounters an error in the input program, you need to continue processing the program. You can print out the illegitimate character and continue on next input character.

You submission should include the following files. Please use the standardized names for your files so that we can test your system uniformly.

- The lex definition file for the FP language. The file name should be “FP.lex”.
- Any other files that are part of your program implementing the project.
- A file containing a sample program written in the FP language that you have used to test your scanner program. The file name should be “sample.fp”. If you wish to submit multiple sample files, you can name them “sample1.fp”, “sample2.fp”, etc.
- A makefile file that compiles your program into “cfp.exe”.
- A readme file that explains how to compile and run your program and how to interpret your output. Also, if you have some information that you would like the TA to know, you can also put it in this file.
- The optional Design.doc file that contains the description of some special features of your project that is not specified in the project specification, including all problems your program may have and/or all additional features you implemented.

3 Project PT -- Syntax Analysis of SFP using Parsing Table

In this project, you need to build a parser from scratch, based on the concept you learnt in class. Consider the SFP language. First, build the action table and goto table of the parsing table manually for the SFP language.

In the action table, you have tokens: {, }, l (loop), p (print), =, f (+, *), n (integer and float), x (identifier). You also have $ in the action table.

In the goto table, you have non-terminal symbols P (program), T (statements), S (statement), R (parameter), C (function call). If you need additional symbols, just use additional upper case letters to represent the symbol.

Now, prepare your parsing table as an input in a file named “input.pt”. The action table and goto table should be stored together as one. You should index the rows of the table by the states.
(starting from 1, because we use 0 for error state, use consecutive numbers) and index the columns of the table by the tokens and symbols listed above. Each cell is separated from its neighbor by a tab. A line break separates two rows of the table. The columns should be arranged in the order of the tokens and the symbols listed above. Special state numbers include 0, which is for error state and 9999, which is for accept state.

In the action table, if a shift action should be taken, then the corresponding cell should only contain the state to go to next. If a reduction action should be taken, then the corresponding cell should contain the production rule (not the index of the rule but the rule itself). We use a simple form to represent production rules. For example,

<statement> ::= { loop <identifier> <statements> } can be written as

S\rightarrow\{lxT\}

Build a parser to work with the tokens generated from lex and parse the token string. The parser program should first reads in the parsing table “input.pt” for initialization and then starts parsing. During parsing, the parser program calls the function in lex to return the next token and process the new token accordingly. Also, the parser program should build a parse tree during the parsing process. The SFP program is fed into your parser program via standard-input. Please do not hard code the file name for the SFP program.

After reading in the parse table, print out the table in a nice format. At the end of parsing, you should print the parse tree in the “pre-order” with indentations. Each node of the tree should be printed in one line. If it is a token, then print the original token string. If it is a grammar symbol, then print the symbol only. After printing a tree node, its child nodes should be printed following it with indentation and the indentation should be 2 blank spaces from the parent starting column.

Your program should be able to handle any input grammar with the confined set of symbols and tokens. In the grading process, we may use a different parse table file to test your parser. The set of tokens and symbols in the testing parse table will be the same as those in the original parse table. Your code should have no hard coded parsing logic or patches so that it can process any parse table.

You submission should include the following files. Please use the standardized names for your files so that we can test your system uniformly.

- The lex definition file for the SFP language. The file name should be “SFP.lex”.
- Your parser program, which has to be named “parser.cpp” or any normal language extension if another language is used.
- Any other files that are part of your program implementing the project.
- A makefile file that compiles your program into “parser.exe”.
- The “input.pt” file.
- A file containing a sample program written in the SFP language that you have used to test your parser program. The file name should be “sample.sfp”. If you wish to submit multiple sample files, you can name them “sample1.sfp”, “sample2.sfp”, etc.
- A readme file that explains how to compile and run your program and how to interpret your output. Also, if you have some information that you would like the TA to know, you can also put it in this file.
The optional *Design.doc* file that contains the description of some special features of your project that is not specified in the project specification, including all problems your program may have and/or all additional features you implemented.

4  **Project YCG -- Syntax Analysis and Code Generation using yacc**

The goal of the project is to use yacc to perform code generation, generate the MC code from an FP program. You are to learn another compiler-compiler tool, yacc, for parsing. You need to define the original FP language in yacc definitions and feed it to yacc to generate a parser. The lexer you wrote in Project Lex should be used together with your yacc program (but will require modifications).

First, you need to convert the BNF definition given for FP into an appropriate grammar. You can change the grammar somewhat to make your programming task more effective, but the language should be the same. (The BNF is a simplified version, so the correct language should also consider semantics of FP. For example, the general function specification does not define the number of parameters each function would take, but the semantics does. You can either define the grammar to ensure the correct semantics or check these semantics specifically during code generation.) You also need to write code in the yacc definition file to generate an MC program from the input FP program. Some specific issues and requirements for code generation are discussed in the subsections.

Your code generator should read the input FP program from the standard input, but the output program name should be given as a command line input. For example, if the executable for the yacc based code generator is codegen.exe. Then we may issue a commend

```
bash> codegen.exe sample.mc < sample.fp
```

So the input program is “sample.fp” and the output MC code is stored in “sample.mc”. Do not use hard coded file names.

An MC executor will be provided so that you can execute the MC code you generated from your FP program. The generator can be access at “~ilyen/mcexe”. It invokes several executables compiled on csgrads1, so can only be used at csgrads1, csgrads2, etc. It requires a file name as the command line input (just the file name, without extension). For example, you may issue command

```
bash> ~ilyen/mcexe sample
```

It will take “sample.mc” as input and create an executable “sample.exe”. You only want to try this if your input FP program is correct and you have generated a correct MC program from it. Otherwise, the resulting executable will not work correctly anyway. Note that this MC executor is for fun, not for you to test your code generation process. You have to read the MC code generated from your code generator to ensure the correctness of your project.

Besides generating the MC program, your code should also create and print the symbol table. The specifics about the symbol table will be discussed later.

You submission should include the following files. Please use the standardized names for your files so that we can test your system uniformly.
The lex definition file for the FP language. The file name should be “FP.lex”.

The yacc definition file for the FP language. The file name should be “FP.yacc”.

Any other files that are part of your program implementing the project.

The sample program(s) written in the FP language (extension should be “.fp” to work with “mcexe”).

A makefile file that compiles your program into “cfp.exe”.

A readme file that explains how to compile and run your program and how to interpret your output. Also, if you have some information that you would like the TA to know, you can also put it in this file.

The optional Design.doc file that contains the description of some special features of your project that is not specified in the project specification, including all problems your program may have and/or all additional features you implemented.

4.1 Symbol Table and Different Types of Identifiers

In the code generation process, you need to maintain a symbol table and use the information in code generation. Your symbol table will include entries for constant names, variables names (destination of an assignment statement), and function names. In MC, you will no longer use identifiers. Each identifier will be processed differently.

For each constant, you need to create an entry for it in the symbol table, storing its name and its constant value. During code generation, this value will need to be used, not the name.

For each variable (including temporary variables), you need to compute and assign the memory address when it is encountered the first time. This address should be maintained in your symbol table. When the same variable is used later, you can obtain and use the already assigned address. When you generate code, the memory address (as defined in <M-addr>) should be used, not the identifier. Note that all variables are of length one word. So the address calculation can be done easily (strings can only be constants).

For each function name, you need to compute the index of its first instruction and record that in the symbol table. Later when the function is called, this instruction location will be given to facilitate code generation. Also, the caller information should be recorded to facilitate the code generation for the return process.

At the end of the program, you need to print your symbol table. Your symbol table at this time contains at least the identifiers, their types, and their memory allocation. For a constant, you should have the constant data type (integer, float, string, Boolean) and its value. For each variable, you should have its memory address, number of times it has been defined or used in the code, etc. For function name, you should have the starting address (the index of the first instruction of the function), the number of callers, the last return address, etc. You should print out these data clearly (specifying what you are printing and the value). For each identifier, you can print its table index, its identifier name, its identifier type, and the information listed above (or more). Try to print the data for each identifier in a single row or have proper indentation and labeling to make it clear to read.
4.2 Data Types

In this project we will tentatively ignore the data type of each variable. A variable could be of type integer or float, but for now we assume that it is always a float. We assume that the input FP program has no % operation (if you wish, you can remove % in your lex definition file, but we will need it in next project). Also, we assume that all type rules for FP (given in Section 5.1) are satisfied (assume that the input program follows all typing rules).

4.3 Process Conditional Statements

For if, loop, and while statements, you need to compute which instruction to go to under the corresponding conditional expressions. You may not be able to know the address at the code generation time till the entire if, loop, or while statement is processed. Your program will need to generate instructions in memory so that the missing addresses can be filled when the information becomes available. At the end of parsing, you can then dump the program to the output file. Or you can build a parse tree (or abstract syntax tree) first and then do code generation based on the parse tree so that you have all information in hand. Building parse tree would be preferred because it is necessary in the last project.

4.4 Process Function Definitions and Function Calls

When a function is defined, the index of its first instruction should be stored in the symbol table. When the function is called, the execution flow jumps to this instruction. Before jumping to a function, the index of the instruction the execution flow has to jump back to has to be recorded. It may be best to allocate a memory location to store the return address while processing the function and the memory address can be stored in the symbol table as well. Also, your program should copy the input parameters to the formal arguments of the function before branching to the function. Similarly, it should copy the returned argument of the function to the caller instruction before branching back to the caller statement.

We make a few assumptions about the function calls in FP to ease the code generation task. (1) Assume that there is no recursive function call. In other words, a function will not be invoked more than once at the same time. (2) Assume that the name of the identifiers are globally unique, including the variables in the functions. This constraint is to let you have a simple symbol table implementation. All variables within a function are local to that function. You do not need to consider scoping rules. Note that these two conditions will be removed in the next phase. So you need to consider the next phase while designing your code in this phase.

4.5 Register and Memory

You need to assign memory for each variable. If you have N variables, then you will have to use N memory words, from 0 to N–1 (because all the variables are of the same size, one word). For each variable, its memory address is maintained in the symbol table (or assigned if the address is not yet available). When you generate code, you cannot use the original identifier, you should use the memory address. The memory is expressed by an array: M[addr]. If a variable is assigned
to memory location 18, then your code should reference $M[18]$ for that variable. In real compiler, this is how variables are handled, except that the address computation is far more complex due to various sizes of variables and the use of pointers and the virtual memory mechanism.

Instructions are executed in CPU and they use registers. When generating code, you need to load the memory content to a register. After the computation is done, you may or may not need to store the register content back to memory. You have 10 registers in CPU (pseudo), R0 to R9. You can use them in the generated code. How the registers are used is not important in this project, but will be important in the next project.

5 Project ExYCG -- Extension to YCG

Now you need to consider some improvements to YCG you developed in the previous project. You definitely need to create a parse tree so that some of the steps required in this project can be performed on the parse tree. The required improvements are discussed in the following subsections. In case you cannot finish all three improvement goals, you can implement a subset of them and get partial credits.

You submission requirements are the same as Project YCG.

5.1 Data Types

In this phase, you need to do type inference to determine the type of each variable and also perform type checking. You can follow the type inference algorithm we have discussed to determine the type of each variable. If it is not possible to determine the data type of a variable between integer and float due to the operator overloading, then the variable is of polymorphic type of \{integer, float\}. In an operation, a variable of a polymorphic type is assumed to be of the highest super type it can have.

Some typing rules are listed as follows:

- Constant identifiers cannot be redefined, i.e., \{= constant-identifier \ldots\} is not allowed.
- \{\% integer integer\} and the result is an integer.
- Parameters for math operations, $+$ $-$ $*$ $/$, have to be of type integer or float and have to be type consistent, i.e., \{$+$ $*$ integer integer \ldots\} or \{$+$ $*$ float float \ldots\} and \{$-$ $/$ integer integer\} or \{$-$ $/$ float float\}. The result of these operations should be consistent with the parameters.
- Parameters for the assignment operation = can only be of type integer or float and they have to be type consistent, i.e., \{= float float\} or \{= integer integer\}.
- The input and return parameters of a function can only be of type integer or float.
- In a function call, the numbers of formal and actual parameters should be equal. Also, the actual parameter can be of the same type or a subtype of the formal parameter.
- The two parameters for comparison operators, $==$ $>$ $<$ $>$= $<=$ $!=$, can only be of type integer or float and they should be of the same type. The result of a comparison operation is of type Boolean (but it is irrelevant since it cannot be used anywhere).
- The identifier used for loop control in the loop statement has to be of type integer.
Note that each constant has its own specific type and when used in various functions/operations, their types should be checked to ensure the conformance of type rules. String type is restricted to constant definition and in print statement, nowhere else. Boolean type is restricted to constant definition and in expressions, nowhere else.

While performing type inference, you need to decide whether there are type errors. If so, you can report all the type errors and involved variables. You need to finish processing the input FP program and report all type errors. In case there are type errors, you do not need to continue with MC code generation, but you do need to continue processing the FP code to identify all type errors.

In MC there is no type definition and in the executor all variables are assumed to be of type float. So you need to maintain the typing of the variables and decide what true operation should be performed. More precisely, you need to code in MC properly to achieve % and / operations in integer data type (use the trunc instruction). For example, if the parameters for / operation are integers, then you need to generate code in MC to simulate integer type /. (In a real system, integer and float data have different representations and integer and float operations have separate instructions. Compiler is supposed to convert the data for proper storage and select the proper instructions for correct computation.)

When you print the symbol table, data types of variables now become an important attribute to be printed (in addition to what you have printed earlier).

5.2 Process Function Definitions and Function Calls

Since there may be recursive calls, it is now necessary for you to maintain a call stack. Each function call should have a record in the call stack. When processing a function definition, you need to define temporary memory locations which is relative to the start memory location for the data in the function. When processing function call, a starting address (for function data) should be stored and added to the relative memory location to obtain the true memory location. You need to generate MC code so that this feature can be achieved.

5.3 Optimization

You need to perform liveliness analysis based on your parse tree. After liveliness analysis, you should perform dead code elimination and then create the MC code according to the parse tree with dead code eliminated.

After liveliness analysis, before dead code elimination, you need to print out the live variables before each definition (assignment statement). You can print out in one line the variable being defined and the Live set before the definition. Please clearly label your printout. For each dead code being eliminated, you need to print it out. Properly label your output.

Also, you need to perform register allocation to determine the most appropriate number of registers to use for the input program and assign registers for code generation.
Liveliness analysis are performed within each function. In other words, both dead code elimination and register assignment are performed function by function (main is also a function).