

# 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 language you need to use to implement the compiler is C++ and the languages defined by lex and yacc. In the following, we first define two new languages by specifying their syntax and tokens. One of the languages is the machine code, MC. The other is a high level language, FP, which has the functional languages' syntax, but has the procedural language characteristics. The compiler you are constructing is to translate the FP language to the MC language. After the language definitions, the projects you need to work on for the course are specified.

Here are some general rules about the projects. First, you are responsible for generating your own testing programs and test your project thoroughly. We will use a different set of input for testing. Second, for each required output, it is your responsibility to print it in an easy to read format and eliminate debugging output. Third, 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. Fourth, the standard platform for this project is the university Sun systems, such as *apache*. You should make sure that your program runs on these platforms.

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 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. According to the problems, appropriate point deductions will apply in these situations.

## 1 Language Definitions for MC

### 1.1 Syntax Definitions

```
<program> ::= <stm> <program> | <stmt>
<stmt> ::= <label> <instruction> ; | <instruction> ;
<instruction> ::= <load-instruction> | <store-instruction> |
                <add-instruction> | <mul-instruction> |
                <if-instruction> | <test-instruction> |
                <goto-instruction> | <print-instruction>

<load-instruction> ::= b <register> <M-addr>
<store-instruction> ::= addin <register> <M-addr>
<add-instruction> ::= add <register> <parameter>
<mul-instruction> ::= m <register> <parameter>
<goto-instruction> ::= go <label>
<if-instruction> ::= if <register> <label>
<test-instruction> ::= <test-operator> <register> <parameter>
<read-instruction> ::= g <register>+
<print-instruction> ::= ad <parameter>+
```

<parameter> ::= <register> | <M-addr> | <number>

<load-instruction> loads (or brings) a memory content <M-addr> to a register <register>.  
<store-instruction> stores (or adds) a register content <register> to a memory slot <M-addr>.  
<add-instruction> computes <register> + <parameter> and stores the result back to <register>.  
<mul-instruction> computes <register> \* <parameter> and stores the result back to <register>.  
<goto-instruction> simply causes the program flow to go to the instruction labeled by <label>.  
<if-instruction> tests if <register> content is positive (true). If so, then the program flow should go to the instruction labeled by <label>. If not, then it continues to the next instruction.  
<test-instruction> tests whether <register> is = or > or >= the parameter <parameter>. If true then <register> is set to 1; otherwise, <register> is set to 0.  
<read-instruction> reads (gets) from the standard input into one or more registers. It can read multiple inputs for multiple registers. When there are multiple inputs, they should be separated by spaces.  
<print-instruction> prints (advertises) the parameters following it. Each parameter should be separated by a tab from the next during printing.

## 1.2 Token Definitions

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 in regular expressions.

<label>'s regular expression is (a | b)+:

<register>'s regular expression is R[0-9]

<M-addr>'s regular expression is (a | b)+, but excluding the keywords

<number>'s regular expression is ( (- | ε)[1-9][0-9]\* ) | 0

<test-operator>'s regular expression is = | > | >=

Note that white space characters, including space, tab, and new-line, are allowed in the input string for convenience, but they are not required and are not a part of the language and should be **ignored**. Please also note that the MC language is case sensitive.

## 1.3 New Syntax Definitions

<program> ::= <stm> <program> | <stmt>

<stmt> ::= <label> <instruction> ; | <instruction> ;

<instruction> ::= <load-instruction> | <store-instruction> |  
                  <add-instruction> | <mul-instruction> |  
                  <if-instruction> | <test-instruction> |  
                  <goto-instruction> | <print-instruction>

<load-instruction> ::= load <register> <M-addr>

<store-instruction> ::= store <register> <M-addr>

<add-instruction> ::= add <register> <parameter>

<mul-instruction> ::= mul <register> <parameter>

<goto-instruction> ::= goto <label>

<if-instruction> ::= if <register> <label>

<test-instruction> ::= <test-operator> <register> <parameter> <parameter>

```

<read-instruction> ::= get <register>+
<print-instruction> ::= print <parameter>+

<parameter> ::= <register> | <M-addr> | <number>

```

Also, <M-addr>'s regular expression is now changed to [a-z]+, but excluding the keywords.

## 1.4 An Example MC Program

```

// This is not a part of the program, just to clarify
// R1 is for storing the temporary and final results for computing N!
// R2 is a constant and temporary variable for comparison purpose
// R3 is the read in value and will be decreased during computation
    mR10;add R1 1;
    mR20;add R2 1;      g
                        R3;
bbb:  >= R2 R3;
      if R2 aaa;
      m R1 R3;      add R3 -1;
      m R2 0; add R2 1;
      go bbb;
aaa:
ad R0;

```

## 2 Language Definitions for FP

### 2.1 Syntax Definitions

```

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

<constant-definitions> ::= <constant-definition>*
<constant-definition> ::= { <constant-name> <number> }
<constant-name> ::= <identifier>

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

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

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

```

```

<parameters> ::= <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.

## 2.2 Definitions for the Remaining Tokens

The definitions for <identifier> and <number> in FP are the same as <M-addr> and <number> in MC, respectively. 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.

To ensure the correctness in processing the input FP-program, you need to consider white space as well. Treat the white space the same way as in other high level languages such as C++ and Java, **skip** them.

## 2.3 Specifications of Some Semantics

Each of the predefined comparison operations takes two parameters and the corresponding functionality follows the conventional definition. If it is not clear to you, please discuss with me in class or during my office hours. The predefined functions + and \* can take one or more parameters and return the sum and product of them, respectively. For example, {+ A B C} returns A + B + C and {\* A} returns A. The predefined function “print” takes one or more parameters. It prints the value of each parameter and returns the first parameter. The operation “read” takes one or more identifiers. It simply reads a value from the standard input for each identifier.

The loop statement takes an <identifier> to indicate the number of iterations for executing the following <statements>. The identifier can only be an integer or real number. If it is a real number, then we take **floor** to get the integer number as the number of iterations.

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.

## 2.4 An Example FP program

```

CONSTANTS
{ aa -1 }
{ bb 1 }

```

```

MAIN
{ = b bb }
{ read a }
{ while { > a bb }
  { = b { * b a } }
  { = a { + a aa } }
}
{ print b }

```

### 3 Project DFA -- Lexical Analysis of MC using DFA

First, build a DFA manually for token recognition for the MC language. Then, key in your DFA as a table in to the file named “**input.dfa**”. You should index the rows of the table by the states in the DFA and index the columns of the table by the alphabets. To ease the grading process, you are required to name each state using a natural number. **One additional column should be added to the table in addition to the regular DFA table. For each state, the last column provides the token name in the case that the token is accepted (otherwise, indicating that it is a non-accepting state).**

Next, build a scanner to recognize tokens for the input MC programs. The MC program is fed into your scanner program via standard-input. **The input string terminates by end-of-file.** Please do not assume any file name for the MC program. The scanner program should first reads in “input.dfa” for initialization and then starts token recognition accordingly.

After reading in the table in “input.dfa”, your program should print out the table in a nice format. After recognizing each token, your program should print out the token type and the token string. Token types include <label>, <register>, <M-addr>, <number>, <test-operator>, and <keywords>. Even though <keywords> is not defined in the language, you can use it as a token type for the instruction operators such as add, mul, etc.

When an <M-addr> is recognized, the scanner should insert it to the symbol table. Maintain your symbol table in the order that the identifiers first appear in the program. In the end, the scanner should print out the symbol table in the order specified above.

Do not forget the first match and longest match rules during DFA construction and scanner coding.

In the grading process, we may go in your “input.dfa” file and make changes to test your scanner functionality. Your code should have no hard coded token recognition logic or patches.

You need to electronically submit your project **before** midnight of the due date (check the web page for due dates). Follow the instruction below for submission.

- Login to one of the university Sun platforms.
- Go to the directory that *only* contains:
  - ✓ Your scanner program, which has to be named “**scanner.cpp**”.
  - ✓ **A makefile for compiling your scanner program.**
  - ✓ The “**input.dfa**” file.

- ✓ A file containing a sample program written in the MC language that you have used to test your scanner program. The file name should be “**sample.mc**”. If you wish to submit multiple sample files, you can name them “**sample1.mc**”, “**sample2.mc**”, 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 **optinal** *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.
- Issue the command *~ilyen/handin/handin.compiler*. This command has to be issued in the directory that contains the files listed above.
- After submitting your project, do not delete or modify your code. Just in case something is wrong, you will have a chance to resubmit your program files with the original dates on them.

#### 4 Bonus Project – Executor for MC Programs

In this bonus project, you are to reverse an MC program to a C++ program. Instead of printing tokens, you should change your scanner program in Project #1 in several ways. First it should record the tokens of the current instruction till a semicolon is encountered (end of instruction). At the end of the current instruction, the scanner should convert the instruction to a C++ statement and print it out to the standard output.

Second, you should make use of the symbol table created at the lexical analysis phase to help with variable declarations. A new identifier should be declared before it is used. You can simplify the task by defining all variables to be “float”.

Third, for the read and print instructions, you can add your own rules to decide how the input should be prompted and how the output should be formatted. For example, for the read instruction, you may want to add a leading statement, “Please input 3 float numbers: ”, to request for multiple inputs.

Fourth, you need to generate a program body for the output C++ code. The program body may include some potential `#include` definitions for I/O. To simplify the process, you only need to define the `main()` body and use a single class.

The C++ code generated by your scanner program should be executable and be able to generate correct output as how the original MC program is designed to do.

Grading for the bonus project will be more strict. Your program has to work fully in order to get the bonus credit. You will need to work with the TA to demonstrate your working program. The TA may ask you to make some specific changes on the spot to ensure that the program is truly working.

## 5 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*, instead manually constructed DFA, for this purpose. 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 a symbol table as well as print the tokens (token type and the original token string) appropriately. You should design the symbol table flexibly such that you can easily add new fields for the symbols in the future.

You need to electronically submit your project **before** midnight of the due date (check the web page for due dates). Follow the instruction below for submission.

- Login to one of the university Sun platforms.
- Go to the directory that *only* contains:
  - ✓ The *lex* definition file for the MC language. The file name should be “**FP.lex**”.
  - ✓ A **makefile** for invoking *lex* and then compiling your program generated by *lex*.
  - ✓ 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 *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 **optinal** *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.
- Issue the command `~ilyen/handin/handin.compiler`. This command has to be issued in the directory that contains the files listed above.
- After submitting your project, do not delete or modify your code. Just in case something is wrong, you will have a chance to resubmit your program files with the original dates on them.

## 6 Project PT -- Syntax Analysis of Simplified FP using Parsing Table

Here we consider a simplified version of the FP language, SPF. The specification for SFP is given in the following.

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

First, build a **parsing table** manually for the SFP language. To make the derivation easier, you may want to identify all the terminals and nonterminals and assign symbolic names for them, instead of using the full names in the BNF specification. Then, key in your parsing table and store it in a file named **"input.parse"**. You should index the rows of the table by the states and index the columns of the table by the tokens. To ease the grading process, you are required to name each state using a natural number.

Second, simplify your lex definition file created in "Project Lex" by removing the tokens that are not in the SFP. Use the new definition file to generate the scanner program for processing SFP.

Third, build a parser to work with the scanner generated from lex and parse the token string. The parser program should first reads in "input.parse" for initialization and then starts parsing. During parsing, the parser program calls the scanner 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 FP program is fed into your parser program via standard-input. Please do not assume any file name for the FP 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 **"in-fix"** order with indentations. Each node of the tree should be printed on one line, start with the original token string and followed by the role of the node, such as "second <parameter> of a <parameters>", "<parameters> of a function call", "loop of a loop statement", "<identifier> of a = statement", etc., and the. 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.

You need to electronically submit your project **before** midnight of the due date (check the web page for due dates). Follow the instruction below for submission.

- Login to one of the university Sun platforms.
- Go to the directory that **only** contains:
  - ✓ 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"**.
  - ✓ **A makefile for invoking lex and then compiling your parser program.**
  - ✓ The **"input.parse"** file.
  - ✓ A file containing a sample program written in the MC language that you have used to test your parser program. The file name should be **"sample.mc"**. If you wish to submit multiple sample files, you can name them **"sample1.mc"**, **"sample2.mc"**, 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 **optinal** *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.
- Issue the command `~ilyen/handin/handin.compiler`. This command has to be issued in the directory that contains the files listed above.
- After submitting your project, do not delete or modify your code. Just in case something is wrong, you will have a chance to resubmit your program files with the original dates on them.

## 7 Project Yacc -- Syntax Analysis of FP using yacc

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. Your definitions should follow the BNF given earlier and should be defined on top of the tokens you defined in Project #2.

You need to write code in the yacc definition file to generate an MC program from the input FC program. Most of the code generation task is relatively simple, generating the corresponding MC instruction(s) for each statement. However, for the function calls, the code generation process is more complicated. Your program should copy the input parameters to the formal arguments of the function before branching to the function. Also, it should copy the returned argument of the function to the caller statement before branching back to the caller statement. Moreover, to place the MC code for functions properly, you may not want to directly print their code segments to the output. You can keep the code segment for a function in a temporary file first. After finishing processing the entire program, you can then read back these files and print them at the end of the output.

The input FC program should be read from the standard input and the output MC program should be printed to the standard output. Do not use hard coded file names.

You also need to maintain the symbol table to help with the code generation process. In a real compiler, symbol table is very critical in code generation. Memory locations (in terms of offsets) are allocated to each variable in the symbol table (size is computed based on its type) and the instruction should access the correct memory address. In our case, we still use the variable name so the translation is much easier.

We make several assumptions about the function calls in FP to ease the code generation task.

- (1) Assume that there is no recursive function call.
  - (2) Assume that a function will only be called once in the entire program.
  - (3) Assume that the scope of all variables are global, including the variables in the functions.
- Thus, when generating code for a function call in FP, you simply copy the actual parameter to formal parameter and jump to the function for execution. Note that the return address can be uniquely determined due to assumption (2). So after execution of a function, you can simply jump back to a predetermined label. You do need to copy back the return value properly.

Below are the examples of the input FP code and output MC code.

```
CONSTANTS
{ neg -1 }
{ one 1 }

FUNCTION
{ factorial n return fac
  { = fac one }
  { while { > n one }
    { = fac { * fac n } }
    { = n { + n neg } }
  }
}
```

```

MAIN
{ read nn }
{ = out { factorial nn } }
{ = out { print out } }

```

The corresponding MC code generated from the FP code above is given below.

```

// set constant neg = -1 and one = 1
mul R0 0 ; add R0 -1; store R0 neg;
mul R0 0 ; add R0 1; store R0 one;
// begin to execute the main function
goto main;

// function factorial
factorial:
    load R0 one; store R0 fac;
tempb:   load R0 n; load R1 one; >= R1 R1 R0; if R1 tempa;;
        load R0 fac; mul R0 n; store R0 fac;
        load R0 n; add R0 neg; store R0 n;
        goto tempb;;
tempa:   goto retfactorial;

// main
main:    get R0; store R0 nn;
        load R1 nn; store R1 n;
        goto factorial;
retfactorial:
        load R0 fac; store R0 out;
        print out;
        load R0 out; store R0 out;

```

An MC executor will be provided so that you can execute the MC code you generated from your FP program. In case you have completed the Bonus Project, you can use your own executor to execute the MC code generated from your code generator.

You need to electronically submit your program **before** midnight of the due date (check the web page for due dates). Follow the instruction below for submission.

- Login to one of the university Sun platforms.
- Go to the directory that **only** contains:
  - ✓ 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**”.
  - ✓ A **makefile** for invoking lex, then invoking yacc, then compiling your program generated by lex and yacc.
  - ✓ The sample program(s) written in the FP language.
  - ✓ 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 **optinal** *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.
- Issue the command `~ilyen/handin/handin.compiler`. This command has to be issued in the directory that contains the files listed above.

- After submitting your project, do not delete or modify your code. Just in case something is wrong, you will have a chance to resubmit your program files with the original dates on them.

## **8 Grade Distribution**

Project DFA	20%
Project Lex	15%
Project PT	30%
Project Yacc	35%
Bonus Project	15%