

# CS 6353 Compiler Construction – Project Assignments

In this project, you need to implement a compiler for a language defined in this handout. The programming language you need to use is C or C++ (and the language defined by the corresponding tools). The project includes three phases, lexical analysis, syntax analysis, and code generation. In the following, we first define the language syntax and tokens. The definitions are given in BNF form. After the language definition, the three phases of the project are specified. The first two phases are mandatory. The third phase is a bonus project. It gives 20 points maximal toward your final project grade.

## 1 Language Definitions

### 1.1 Syntax Definitions

```
program ::=
    "{" "Program" program-name function-definitions statements "}"
program-name ::= identifier

function-definitions ::= (function-definition)*
function-definition ::=
    "{" "Function" function-name arguments statements "return" return-arg "}"
function-name ::= identifier
arguments ::= (argument)*
argument ::= identifier
return-arg ::= identifier | ε

statements ::= (statement)+
statement ::= assignment-stmt | function-call | if-stmt | while-stmt

assignment-stmt ::= "{" "=" identifier parameter "}"
function-call ::= "{" function-name parameters "}" | "{" predefined-function parameters "}"
predefined-function ::= "+" | "-" | "*" | "/" | "%" | "print"
parameters ::= (parameter)*
parameter ::= function-call | identifier | number | character-string | Boolean
number ::= integer | float

if-stmt ::= "{" "if" expression "then" statements "else" statements "}"
while-stmt ::= "{" "while" expression "do" statements "}"
expression ::= "{" comparison-operator parameter parameter "}" |
    "{" Boolean-operator expression expression "}" | Boolean
comparison-operator ::= "==" | ">" | "<" | ">=" | "<=" | "!="
Boolean-operator ::= "or" | "and"
```

To make the later references easier, let's call this language the FP language (similar to functional language syntax, but has the procedural language characteristics).

## 1.2 Definitions for the Remaining Tokens

An identifier has the form `[a-z, A-Z][a-z, A-Z, 0-9]*`, but its length has to be between 1 to 6 characters. Confining the number of characters in an identifier is not a good language design feature, but this is for you to practice a different definition. You have to use regular expression to achieve the definition of the length constraint (you are not allowed to use the length feature in lex to confine the length).

An integer can have 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).

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. 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, \, \ ]+`. For example, `(I like lex and yacc)` is a character string. We use `\` to represent a new line, i.e., `\n` in C. The character strings are mainly used in the print function. We may also define functions that uses character strings as parameters.

A Boolean can only be `"T"` or `"F"`, where T represents true and F represents false.

## 1.3 Definitions of the Predefined Functions

Each of the predefined comparison operations and Boolean operators 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 `%` also have the conventional meanings. `+`, `*`, `-`, and `/` can take two or more parameters. For example, `{* A B C}` implies `A * B * C`. Similarly, `-` and `/` can also take two or more parameters. For example, `{/ A B C}` implies `A / B / C` and `{- A B C}` implies `A - B - C`. The `%` function can only take two parameters and `{% A B}` means `A % B` (`A mod B`).

The predefine function `"print"` takes one or more parameters. It simply prints the value of each parameter. Whenever a character string `()` is encountered, a new line should be printed.

Although there are functions that take a fixed number of parameters, it is not your job to check the number of parameters (not in the lexical and syntax analysis phases). Also, it is not necessary to check whether the number of arguments and number of parameters do match (not in the lexical and syntax analysis phases). You only need to follow the general rules defined in the syntax of the language.

## 1.4 White Space

To ensure the correctness in processing the input FP-program, you need to consider white space as well. White space characters include space, tab, and new-line. Treat the white space the same way as in other high level languages such as C++ and Java (skip white space). Note that space in character string should not be skipped.

## 1.5 Some Remarks

Note that we do not consider the language typing rules. Thus, there is no need to predefine variables and your program does not need to handle mismatched types.

The FP language is case sensitive.

Here is an example program using the FP language.

```
{Program Sample1
  {Function facto VAL
    {if {< VAL 0 }
      then {= retVal -1}
      else {= retVal 1}
      {while {> VAL 0} do
        {= retVal {* retVal VAL}}
        {= VAL {- VAL 1}}
      }
    }
  return retVal
}
{print {facto 999}}
```

## 2 First Project -- Lexical Analysis Using lex

Define the tokens in the FP language in lex definitions and feed it to *lex* to generate a scanner (lexer). Then, use the sample program as well as other testing programs written in FP to test your scanner. Note that it is your responsibility to convert the BNF (or verbal) definitions to lex definitions. The tokens in FP include:

The set of keywords: Program, Function, return, if, then, else, while, do, or, and, print.

The set of special symbols: {, }, =, +, -, \*, /, %, ==, >, <, >=, <=, !=.

The set of other tokens: identifier, integer, float, character-string, Boolean.

It is also your responsibility to insert appropriate statements (actions) in your lex definition file so that the lexer (created by *lex*) will generate a symbol table and print the output token names appropriately. You should design the symbol table such that it contains the necessary fields for this as well as the future projects.

The token names should be clearly printed, one on each line, in the correct order. The names of the tokens should conform to those given above. The original symbols for the predefined

functions and the comparison-operators should be used as the token names (such as  $\geq$ ). The keywords itself should be used as token names (such as while). For “the other tokens”, the specific names: identifier, integer, float, character-string, and Boolean should be used. Note that for “the other tokens”, you should output the token names as well as the “texts” you obtained (in the same line). If you do not keep track of the original texts, such as the identifier’s name, the integer’s value, the actual character string, etc., then, the information will be lost. For the

The symbol table should be printed as well. You need to print out the content of the symbol table, one entry on a line. (In this phase, you only have the identifier names).

The standard platform for this project is the university Sun system, including the servers and workstations such as *apache*. You should make sure that your program runs on these platforms.

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 FP language. The file name should be “FP.l”.
  - ✓ A file containing a sample program written in FP language that you have tested with the scanner generated from your lex definition file. The file name should be “sample.fp”. If you have multiple sample files, you can name them “sample1.fp”, “sample2.fp”, etc.
  - ✓ A *readme* file that explains how to interpret your output from lex (the list of tokens). 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.

You are responsible for generating your own testing programs and test your project thoroughly. We will use a different set of FP programs for testing.

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 given. 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.

### 3 Second Phase -- Syntax Analysis Using yacc

For the second project, you need to define the FP-language in yacc definitions and feed it to yacc to generate a parser. Your definitions should follow the BNF given earlier, but exclude the token

definitions which have already been processed by lex and instead use the token names you defined for the first project. You also need to generate a parse tree for the input FP program. For parse tree generation, you need to write code in the definition file for tree node generation. Also, you need to use a stack to keep track of the tree nodes so that you can link them properly into the correct parse tree. You can either use yacc stack or your own stack for this purpose.

Each node in your parse tree should include the actual symbol of the node, including all those defined in the BNF for the FP language (e.g., +, "Program", argument, return-arg, statement, statements, if-stmt, etc.) For each identifier, instead of giving the actual identifier name, it should simply be a pointer to the symbol table (the index of the symbol table).

You have created a symbol table in the first project. Now you need to make sure you can access it and add more information into it. In this phase, you need to assign identifier type to each identifier. Identifier types include:

program-name

function-name

argument

return-arg

assignment-id: the identifier in the assignment statement and is to be assigned a value

expression-id: appear after comparison operators or Boolean operators

parameter: all other parameters besides those listed above

At the end (best is in the main program), you need to print out two things: the symbol table and the parse tree. For the parse tree, print it in the "**pre-fix**" order with indentations. Each node of the tree should be printed on one line. After you print 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.

For your symbol table, please try to print each identifier in the table on one line. In each line, first print the identifier name, followed by the identifier type(s), then followed by any other information you have put in the table. **If an identifier appears multiple times and has multiple identifier types, then print all the different types (order is not important).**

Please note that in the lexical analysis in Project 1, you may have processed some tokens too early and it may cause incorrect processing of the input program. You may have to revise the lex definition file and push some of the processing to be done at the yacc level. It is your responsibility to find the problems and make the changes.

You can use the sample program as well as other testing programs written in FP to test your parser. Note that it is your responsibility to thoroughly test your program.

The standard platform for this project is the university Sun systems, including the servers and workstations such as *apache*. You should make sure that your program runs on these platforms.

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 FP language. The file name should be “FP.l”.
  - ✓ The yacc definition file for FP language. The file name should be “FP.y”.
  - ✓ A file containing a sample program written in FP language that you have tested with the scanner generated from your lex definition file. The file name should be “sample.fp”. If you have multiple sample files, you can name them “sample1.fp”, “sample2.fp”, etc.
  - ✓ A *readme* file that explains how to interpret your output from lex (the list of tokens). 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.

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 given. 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.

#### 4 Third Phase -- Code Generation (Bonus Project)

In this phase, you should generate C++ or Java code from the input FP program. You can either achieve code generation in one pass (generate code while parsing) or in multiple passes. In the later case, you can use the parse tree you built in Phase 2 and generate code by going through the parse tree.

You need to do some additional works beyond one to one translation. The first and the most difficult task is that you need to declare the variables. You can simplify the task by defining all variables to be “float”. Though this is not a good practice, there is no simple solution to the problem. You can also make use of the symbol table to help with declaration statements generation.

Another problem in the code generation is that the target language is based on the object-oriented model while FP does not. We can simplify the problem by only use a single class and all objects are public.

There are also undefined semantics in the FP language. For example, the parameter passing mechanism is not defined. Here, we assume that the parameter passing is passed by value. For other undefined semantics that may come up, you can make assumptions that they are the same as your target language and document them in your design document.

For the print statement, you can add your own formatting rules to decide how the output is printed.