CSE 326 - Compiler Construction
A Compiler Project
Department of EECS, North South University

Synopsis

As part of the CSE 326 course, you will be designing a compiler to translate a small subset of C language into Intel Assembly language. This complier consists of a lexical analyzer, a parser, a semantic checker and a code generator. Automatic tools such Lex and Yacc will be used extensively for this project.

PART I - Lexical Analyzer

Design a lexical analyzer using the Lex tool to recognize the following subset of C programming language:

Program Structure: The input file to the lexical analyzer will only have two sections. The first is an optional declaration section and second is only one function. No need to handle # directive such as #include, #define, etc.

Lexemes: Here are the lexemes that need to be handled by your lexical analyzer. You should print out an "Invalid character" message to the standard output for all other characters.

- **White spaces** such as spaces, tabs, and newlines will be ignored.
- **Identifiers** are made up of characters A to Z, a to z, 0 to 9, and underscore (_). The name of the identifier must not start with a number or underscore and have no limitation on length.
- **Keywords** that need to be recognized are **int**, **float**, **for**, **do**, **while**, **if**, **else**, **return** and **void**.
- **Special characters** are as follows:

```
    + - * / %
< > == <= >= != !
     ( ) { }
   = ; , && ||
```

- **Constants** of 2 kinds needs to be recognized:
  - Integer, for example 200
  - Floating, for example 2.1314

- **Comments** start with /* and end with */

Symbol Table: You need to implement a symbol table and an interface to access the symbol table as discussed in detail in the class. Think about what kind of data structure to use for the symbol table and come up with an efficient implementation. This symbol table will have to be updated as you continue developing the rest of the compiler modules.

Tasks:

1. Create a scanner.l file implementing the above lexical analyzer
2. Create symbol.c and symbol.h file implementing the symbol table

3. The lexical analyzer should read the input from the keyboard and display the following messages on the screen from the ***main()*** function:
   a. For all valid tokens, print "Line# ?: <Token Type, Token Value>"
   b. For any unmatched pattern, print "Error: Line# ?: Invalid Character"
   c. Print all keywords together from the symbol table, "Keywords: ? ? …"
   d. Print all the identifiers together from the symbol table, "Identifiers: ? ? …"

<u>Deliverables:</u>

1. Project header page with your name, ID and indicate PART I
2. Hard copy of scanner.l, symbol.c and symbol.h file
3. Email soft copy of only the above 3 files (do not include any other files)

PART II - Parser

Use the Yacc tool to generate a parser for this project. The CFG defining the syntax of the subset of C is given below. You are required to modify these grammars, if necessary, to eliminate ambiguity discussed in the class.

```
program               →       external
external              →       declaration external | function

declaration           →       type_name variable_list ;
type_name             →       void | int | float
variable_list         →       variable_list, variable | variable
variable              →       id

function              →       function_header { function_body }
function_header       →       type_name id ( paramerter_list )
paramerter_list       →       paramerter_list, type_name id | void
function_body         →       internal statement_list
internal              →       declaration internal | ϵ

statement_list        →       statement_list, statement | ϵ
statement             →       { statement_list }
                      |       for_statement
                      |       while_statement
                      |       dowhile_statement
                      |       if_statement
                      |       expression ;
                      |       return expression ; | return ;

for_statement         →       for ( expression ; expression ; expression ) statement
while_statement       →       while ( expression ) statement
dowhile_statement     →       do statement while ( expression ) ;
if_statement          →       if ( expression ) statement
                      |       if ( expression ) statement else statement
```

```
expression            →      unary_expression = expression | binary_expression

unary_expression→     unary_operator unary_expression | primary_expression
unary_operator        →      + | - | !
primary_expression    →      id | constant | ( expression )
constant              →      inum | fnum

binary_expression     →      binary_expression binary_operator unary_expression | unary_expression
binary_operator       →      + | - | * | / | %
                      |      < | <= | == | != | > | >=
                      |      && | ||
```

Tasks:

4. Create a parser.y file implementing the above syntax analyzer
5. Make necessary changes to scanner.l, symbol.c and symbol.h from PART I
6. The syntax analyzer should read the input from the keyboard and display parser error messages if there is any syntax error, for example, consider the following input:

```
int count, total;
float average;

float main (void)
{
    int i;
    for (i=0; i<10; i = i + 1) {
            total = total + i;
    }
    count = i;
    average = total / count;
    return average;
}
```

Deliverables:

4. Project header page with your name, ID and indicate PART II
5. Hard copy of scanner.l, parser.y, symbol.c and symbol.h file
6. Email soft copy of only the above 4 files (do not include any other files)

PART III – Semantic Checker

You will write your own C code within the parser.y file to implement an **abstract syntax tree (AST)** which is needed to implement the following semantic checks:

- All data and function type should be strongly enforced
- Same name will not be allowed for data and function
- For arithmetic operations, correct arguments should be checked for unary and binary operators
- Report any semantic errors with appropriate message to standard output

- Perform necessary type conversion during the parsing of the tree

PART IV – Code Generator

Create a separate .c module for this part; you will convert the abstract syntax tree (AST) into Intel assembly code without going through any intermediary translation. Need to address the following:

- Allocation of storage of all variables
- Efficient register allocation
- Allocation of constants
- Generation of Intel assembly codes

Final Deliverables

For the final submission, you need to submit a Word file describing the overall approach, data structure, grammar modifications, etc. This should also include what works and what does not work for the specific components of the compiler. Include Hardcopy of the latest scanner.l, parser.y, symbol.c, symbol.h, and other .c/.h modules you have written manually.

Email a complete listing of all the files used in the compiler along with the word file.

Due Dates

Part I :	Midnight of June 23, 2012
Part II :	Midnight of July 14, 2012
Part III :	Midnight of July 28, 2012
Part IV :	Midnight of August 8, 2012