CASE WESTERN RESERVE UNIVERSITY
Case School of Engineering
Department of Electrical Engineering and Computer Science
EECS 337 Systems Programming (Compiler Design)
Fall 2013
Assignment #4
30 points + 5 extra credit
Due: September 24, 2013

Part 1: Reading and Exercises

Compilers - Principles, Techniques, & Tools, 2nd Edition, Sections 3.4, 3.5, 3.6
Homework Exercises – laboratory and answer the questions at the end of the assignment.

Introduction

In the last assignment you started with an ANSI C scanner (scan.l) and parser (gram.y) then added code to remove C comments. With the +debug flag you printed out the TOKEN stream with ATTRIBUTES to the console. In this assignment you copy over the compiler from Assignment 3 to a new directory (hw04) and implement a lexical-analyzer to store the attributes for identifiers, constants and string literals. You add a routine to the exit function to print the saved attribute table using a new flag (+symbol).

In a future assignment you add actions to the parser to process declarations into the symbol table using the information from the attribute table. The attribute table is a collection of text strings for identifiers, constants and string literals. All these strings never change during the many phases of the ANSI C compiler so there is no need to duplicate the strings. For each string, if you find the same attribute, then return that attribute else create a new entry. For constants you need to know the format (character, decimal, hexadecimal, octal, floating point) for encoding the strings into a value. Encode the values into the largest integer or floating point variable for your machine architecture. In the future, you will use these values in the compiler instead of using the strings (1.23E-4 instead of “1.23E-4”). The lex scanner expressions allow you to know the format so you can encode the value into the attribute table using the correct format type (sscanf). You verify the encoded value when you print out the attribute table.

ANSI C Scanner

The scanner (scan.l) uses a number of global variables for passing data to the parser (gram.y).

```
extern char   *yytext;           /* holds the ascii characters from scanner */
extern int    yyleng;            /* defines how many characters in yytext */
extern YYSTYPE yylval;           /* defines attribute data structure */
```

The global variables hold the attribute information for identifiers, constants and string literals. Constants are defined in the ansi_c compiler in the scan.l file. Because there are a number of valid characters for each constant a set of macros are defined at the top of the file before the first `%{`. These macros are used to handle a range of characters and make the lexical analyzer regular expressions more readable. The macros are defined below.

```
D                   [0-9]
L                   [a-zA-Z_]
H                   [a-fA-F0-9]
E                   [Ee][+-]?{D}+
FS                  (f|F|l|L)
IS                  (u|U|l|L)*
```

D is defined as a digit from 0 to 9. L is defined as a letter from a to z, A to Z and the underscore (_) character. The L macro is used to define the start of an identifier. H is defined as a hexadecimal digit from a to f, A to F and the digits 0 to 9. E is defined as the exponent part of a floating-point number. FS and IS are modifiers for defining long, float or unsigned constants. The

others macros are used to define constants for decimal, hexadecimal, octal and floating-point numbers. You will see that octal numbers use the D macro and static checking is required.
The scanner uses the macros to define the input patterns and a decimal number is defined in the patterns/actions section (between the `%%` and `%%)` as:

```
{D}+{IS}?
```

This means a decimal number is one or more digits followed by zero or one IS characters. A number without an IS letter is four bytes long (sizeof(123)). If the u or U letter is appended then the number is unsigned. When the L or l is added to the number then it is eight bytes long (sizeof(123l)). Notice the + or – sign is not included in the definition and is treated as a unary operator inside the ansi_c productions. If a constant is used in the program with a minus sign then that value can be out of range. This type error is handled during run time (dynamic checking).

To encode the ASCII digits for decimal, we use a function that returns an index or pointer from the attribute table. In this assignment you edit the lines in scan.l that return a CONSTANT to load the yylval with the attribute index or pointer. Notice that yytext and yyleng are passed into the routine even though they are global variables. This allows the encoder function to be reused in other applications where the buffer and length are not global variables. For example one way to implement the code would look like:

```
{D}+{IS}? { count(CONSTANT); yylval.token = CONSTANT; yylval.index = attribute(
CONSTANT, yytext, yyleng, FORMAT_DECIMAL); return(CONSTANT); }
```

We would like to store the string that makes up the constant and then encode the constant value. We write the encoder function using a type native to the format string in the sscanf function. Below is an example of an attribute function prototype.

```
/*
 *     scanner attribute register function
 *     return an index into the attribute table (static)
 */
int    attribute( int token, char *buffer, unsigned int length, int format)
{
/*
 *     find the same string and return the index
 */

/*
 *     else update to the next attribute field
 */

/*
 *     encode the constant string into a value
 */

       return data.index;
};
```

In the ansi_c scanner a hexadecimal number is defined as:

```
0[xX]{H}+{IS}?
```

This means a hexadecimal number starts with a 0 followed by a lower or upper case x, has one or more hexadecimal digits and is followed by zero or one IS character. Example 0x0fu. Notice only hexadecimal digits can be used but there is no limit on the number of digits. To encode a hexadecimal number use the %Lx format string in the encode statement.
Example: `unsigned long long x; sscanf( buffer, "%Lx", &x);`

An octal number is defined as:

```
0{D}+{IS}?
```

This means an octal number starts with a 0 followed by a one of more digits followed by zero or one IS character. Example: 0377u. Notice that it uses decimal numbers and some are not in the range of octal numbers. Example 0389. This could be corrected by adding a macro to the top with the letter O and then change the patterns to:

```
O [0-7]
0{O}+{IS}?
```

This seems to make sense but there is a trade-off with this approach. If non-octal digitals are allowed then they can be caught in the scanner audit routine and a warning message can be generated. If the digits are not allowed then the scanner will break the constant into two parts (03, 89) and a syntax error will occur from the parser (CONSTANT CONSTANT). To encode an octal number use the %Lo format string in the encode statement.
Example: `unsigned long long o; sscanf( buffer, "%Lo", &o);`

A floating point number has three lexical regular expressions and is defined as:

```
{D}+{E}{FS}?
{D}*"."{D}+({E})?{FS}?
{D}+"."{D}*({E})?{FS}?
```

In all of these lex patterns there is either a digit with exponent or a digit with a decimal point in the string (.1, 1., 1e0). To encode a floating point number use the %Lf format string in the encode statement. Another way, a double is encoded with a %g and an exponent only floating point number as a %e.
Example: `long double f; sscanf( buffer, "%Lf", &f);`

A char is defined as:

```
'(\\.|[^\\'])+'
```

In this pattern a character is defined as starting with a single quote, a character sequence and ending with another single quote. A printable ASCII character normally starts at space and ends at '~' but any 8 bit value can be entered. To enter any value in hexadecimal format use: '\xhh' where h is any hexadecimal digit. To enter any value in octal format use: '\nnn' where n is any octal digit. A number of special characters are also defined as follows, Special Characters:

```
'\n'  0x0A newline
'\r'  0x0D cr
'\t'  0x09 tab
'\b'  0x08 backspace
'\\'  0x5C backslash
'\?'  0X3F question mark
'\''  0x27 single quote
'\"'  0x22 double quote
```

Page 143 Lexical Analyzer Program
The scanner on page 143 does not use the same macros as the ansi_c compiler. It defines regular expressions between the %} and %% section of the lex file. They are shown below:

```
/* regular definitions */
delim    [ \t\n]
ws       {delim}+
letter   [A-Za-z]
digit    [0-9]
id       {letter}({letter}|{digit})*
number   {digit}+(\.{digit}+)?(E[+-]?{digit}+)?
```

Part 2: Laboratory
Make a directory on your computer in your EECS337 directory and call it hw04.
mkdir EECS337/caseid/hw04/ ; where caseid is YOUR Case ID, enter all in lower case

Download a copy of: hw04_caseid.tar file to the hw04 directory from http://blackboard.case.edu/ in the assignment area. From a console window, change directory to the hw04 directory.
cd EECS337/caseid/hw04/

To untar the tar file type the command below.
tar xvf hw04_caseid.tar

The following files will be created in the current working directory.
Makefile
hw04_test.sh
attribute.c

Copy the previous compiler program by using the following commands:
cp ../hw03/main.c .
cp ../hw03/scan.l .
cp ../hw03/gram.y .
cp ../hw03/yystype.h .
cp ../hw03/tokens2.c .

Part 3: Laboratory Assignment Update
Edit the main.c file and add code inside the main_exit function to print the attribute table when the symbol flag is set. Condition the code with the YYDEBUG variable for future optimization.
```
/*
 *     print the attribute table
 */
#ifdef YYDEBUG
       if( IS_FLAGS_SYMBOL( data.flags))
       {
              print_attribute_table();
       }
#endif
```

Add the +symbol and –symbol flags to the command line function.
```
              else if( !strncmp( command, "+symbol", strlen( command)))
              {
                     SET_FLAGS_SYMBOL( data.flags);
                     return;
              }
              else if( !strncmp( command, "-symbol", strlen( command)))
              {
                     CLR_FLAGS_SYMBOL( data.flags);
                     return;
              }
```

Change the usage to include the `[[+|-]symbol]` flag. Save the main.c file.

Edit the yystype.h file. First we must define the SYMBOL flag. Inside the DATA structure add the code for the symbol flag where the other flags are defined.
```
#define       FLAGS_SYMBOL  0x0008
#define       IS_FLAGS_SYMBOL(a)   (a & FLAGS_SYMBOL)
#define       SET_FLAGS_SYMBOL(a)  (a |= FLAGS_SYMBOL)
#define       CLR_FLAGS_SYMBOL(a)  (a &= ~FLAGS_SYMBOL)
```

We are going to redefine the YYSTYPE structure from an integer to a structure, so we must do it before the y.tab.h include file. So before the `#include "y.tab.h"` line add your structure definition of the attribute table (ATTRIBUTE) and your definition of YYSTYPE. For example:

```
/*
 *     define an attribute structure
 */
#define       ATTRIBUTE    struct attribute
ATTRIBUTE
{
       int    token;
#define       MAX_BUFFER_SIZE     128
       char   buffer[ MAX_BUFFER_SIZE];  // static buffer
       int    length;
#define       FORMAT_NONE         0
#define       FORMAT_CHAR         1
#define       FORMAT_DECIMAL      2
#define       FORMAT_HEXADECIMAL  3
#define       FORMAT_OCTAL        4
#define       FORMAT_FLOAT        5
       int    format;
};
/*
 *     define yystype
 */
#define YYSTYPE struct yystype
YYSTYPE
{
       int    token;
       int    index;
};
```

We must now add the attribute table to the DATA structure. For example using a static memory model we add the following code. We define the maximum size and use the index variable as the next available attribute field.

```
/*
 *     define the scanner attribute table (static)
 */
#define       MAX_ATTRIBUTES      128
       ATTRIBUTE    attributes[ MAX_ATTRIBUTES];
       unsigned int  index;
```

We are adding a new file (attribute.c) and need to extern the functions in this file. Add the function prototypes below at the end of the file. You may change these functions. Save the yystype.h file.

```
/*
 *     external variables and functions from attribute.c
 */
extern void   print_attribute( int index);
extern void   print_attribute_table( void);
extern int attribute( int token, char *buffer, unsigned int length, int format);
```

Edit the scan.l file. The first thing we want to do is load the TOKEN into the yylval.token field for **every** lex expression. For example change "auto" to the following code and repeat for each lex expression.

```
"auto"              { count(AUTO); yylval.token = AUTO; return(AUTO); }
```

For identifier, constant and string literal add the calls to the attribute function. Be sure to change the TOKEN and format arguments to match the expressions. When finished save the scan.l file.

```
{L}({L}|{D})*       { count(IDENTIFIER); yylval.token = IDENTIFIER;
yylval.index = attribute( IDENTIFIER, yytext, yyleng, FORMAT_NONE);
return(check_type()); }
```

Edit the attribute.c file. Design, implement and debug the three functions below using the data structures provided or change them to meet your needs. When you are done then save the file.

```
void   print_attribute( int index);
void   print_attribute_table( void);
int attribute( int token, char *buffer, unsigned int length, int format);
```

Build the ansi_c program by using the commands: **make clean** and **make**

Test the ansi_c program by using each of the commands:
./ansi_c +symbol < ../hw02/Code_1_6_1.c
./ansi_c +symbol < ../hw02/Code_1_6_2.c
./ansi_c +symbol < ../hw02/Code_1_6_4.cpp

Your output for the first test file should look something like.

```
for caseid start time: Mon Sep 16 16:48:51 2013
attribute table:
token: IDENTIFIER     buffer: main   length: 5      format: NONE
token: IDENTIFIER     buffer: argc   length: 5      format: NONE
token: IDENTIFIER     buffer: argv   length: 5      format: NONE
token: IDENTIFIER     buffer: w      length: 2      format: NONE
token: IDENTIFIER     buffer: x      length: 2      format: NONE
token: IDENTIFIER     buffer: y      length: 2      format: NONE
token: IDENTIFIER     buffer: z      length: 2      format: NONE
token: IDENTIFIER     buffer: i      length: 2      format: NONE
token: CONSTANT       buffer: 4      length: 2      format: DECIMAL       decimal: 4
token: IDENTIFIER     buffer: j      length: 2      format: NONE
token: CONSTANT       buffer: 5      length: 2      format: DECIMAL       decimal: 5
token: CONSTANT       buffer: 7      length: 2      format: DECIMAL       decimal: 7
token: CONSTANT       buffer: 6      length: 2      format: DECIMAL       decimal: 6
token: CONSTANT       buffer: 8      length: 2      format: DECIMAL       decimal: 8
token: IDENTIFIER     buffer: printf length: 7      format: NONE
token: STRING_LITERAL buffer: "w:%d,\tx:%d,\ty:%d,\tz:%d\n" length: 30      format: NONE
token: CONSTANT       buffer: 0      length: 2      format: DECIMAL       decimal: 0
```

Part 4: Laboratory Output Generation
When all your lab assignments have been completed execute the homework script file. Print out your implemented version of attribute.c and the hw04_test.txt output files. Put your name, assignment number and date on it. Answer the questions at the end of the assignment.
./hw04_test.sh &> hw04_test.txt

Your final directory should look like (using your CaseID):
EECS337/caseid/hw04/Makefile
EECS337/caseid/hw04/attribute.c
EECS337/caseid/hw04/gram.y
EECS337/caseid/hw04/hw04_caseid.tar
EECS337/caseid/hw04/hw04_test.sh
EECS337/caseid/hw04/hw04_test.txt
EECS337/caseid/hw04/main.c
EECS337/caseid/hw04/scan.l
EECS337/caseid/hw04/tokens2.c
EECS337/caseid/hw04/yystype.h

Part 5: Extra Credit (5 points)
Change the static memory model to use dynamic memory. Change YYSTYPE from an integer index into the attribute table to return a pointer to ATTRIBUTE *attribute; structure. Add a routine to the exit function to free the dynamic memory. When done, upload **ONLY** your attribute.c file to blackboard in the assignment area. Submit it with the comments **hw04** with your **CaseID** on the title line. The extra credit will be graded on-line and your score will be added to your assignment score.

Part 6: Laboratory Questions

1. Complete the table below for each of the variables and determine the sizeof operator.

ANSI C types and variables	***sizeof(type/var)***
char char1;	
short short1;	
int int1;	
long long1;	
long long long2;	
signed signed1;	
unsigned unsigned1;	
float float1;	
double double1	
long double double2;	
const const1;	
volatile volatile;	
void void1;	
struct structS { int int1; int int2; }; struct structS structS1;	
union unionS { int int1; int int2; }; union unions unionS1;	
enum Boolean { NO, YES }; enum boolean boolean1;	
typedef unsigned int uint; uint uint1;	
CONSTANTS: hex, octal, decimal, character, floating point	***sizeof(constant)***
0xf	
0xfl	
0xfu	
0xful	
0123	
0123l	
0123u	
0123ul	
123	
123l	
123u	
123ul	
‘a’	
‘\x61’	
‘\141’	
‘aa’	
‘aaa’	
‘aaaa’	
1.2e3f	
1.2e3	
1.2e3l	

2. What basic type is not allowed to be declared but can be used in the sizeof operator?

3. How is the identifier (id) on page 143 of your book different than the ansi_c identifier?

4. List two floating point numbers not supported on page 143 that are accepted by the ansi_c scanner?

5. List two constant types supported by the ansi_c compiler that are not supported on page 143?