CASE WESTERN RESERVE UNIVERSITY
Case School of Engineering
Department of Electrical Engineering and Computer Science
EECS 337 Systems Programming (Compiler Design)
Fall 2013
Assignment #07
30 Points + 5 extra credit
Due: October 15, 2013

Part 1: Reading and Exercises
Compilers - Principles, Techniques, & Tools, 2nd Edition, Sections 6.1, 6.2, 6.3, 6.4
Homework Exercises – problems 6.1.1, 6.1.2 (a,b), 6.4.3 (a,b), 6.4.6 (a,b,c) and laboratory assignment.

Introduction

A compiler normally generates two kinds of intermediate code representation (IR). The first includes trees and the second includes linear representations. Trees (tree data structure) are implemented as parse trees, abstract syntax trees and directed acyclic graphs (DAGs). Linear representations (linked lists) are normally implemented as three address codes (quadruples, triples). Quadruples have an instruction, destination and one or two operands. The destination is normally a temporary operand and not assigned to the identifier until the last step. Triples have an instruction and one or two operands and the destination is implied by its position. The operand is then a pointer to the triples data structure instead of a temporary operand.

The tree structure (dynamic memory) generated from the parser allows for static checking and optimization before converting into a linear representation. A typical algorithm is shown below.

```
tree = yyparse();
tree = static_checking( tree);
tree = tree_optimize( tree);
quad = tree_2_quad( tree);
quad = quad_optimize( quad);
quad_2_target( quad);
```

Some compilers combine these functions into one step and generate the linear representation (dynamic memory) directly from the parser to pass to the code generator. A typical algorithm is shown below. *(note: these algorithms are not real C code fragments)*

```
quad = yyparse() { static_checking( quad); };
quad = quad_optimize( quad);
quad_2_target( quad);
```

An interpreter performed these functions in real time (static memory) and the result is calculated inside the parser at the time of the production reduction (`expr '+' expr { $$ = $1 + $2;}`).

The book defines nine intermediate code representations (three address codes) to implement all programming languages. These can be converted into target instructions for any Von Neumann processor. The first three address codes also apply to an interpreter implementation with identifiers. Consider:

1) **X = Y op Z // binary operator**
2) **X = op Y // unary operator**
3) **X = Y // copy operator**
4) GOTO L
5) IFTRUE X GOTO L and IFFALSE X GOTO L
6) IF X relop Y GOTO L
7) CALL P,N or Y = CALL P,N and RETURN or RETURN Y
8) X = Y[i] or X[i] = Y
9) X = &Y or X = *Y or *X = Y

In the last assignment you implemented the math calculator program (interpreter) from page 295-296 of the book. The program converted input text into constants and operators. The result was calculated and printed to the screen. The operators included unary and binary math and bit wise instructions. Constants were handled as integer or floating-point numbers. For extra credit the constant format (decimal, octal, hexadecimal) were added as an attribute.

You already implemented the first three intermediate code representations for the right side using constant values inside the interpreter. In this assignment you add identifiers and the assignment operator (`left=`) to the parser. You shall use a symbol table to hold the identifiers and values (hint: memory storage for a calculator). On an assignment operation you update the identifier with the result from the parser. The math interpreter shall allow the identifiers to be created without declarations (~~`int`~~ `x = 1`) and allow the type (integer, floating point) to switch based on the result (`x = 1.0`). If you type `x = 1` then the integer result shall be assigned to the variable x and no output shall be generated. If you then type `x + 2`, the identifier x is found in the symbol table and the previous value shall be used to calculate the final result (`3`). The final result shall be printed to the screen. If you type `x` before anything is assigned to it then print just the symbol `x`.

Part 2: Laboratory

From a console window, make a directory on your computer in your EECS337 directory under your Case ID and call it hw07.
mkdir ~/EECS337/caseid/hw07/ ; where caseid is YOUR Case ID, enter all in lower case

Change directory to the hw07 directory.
cd ~/EECS337/caseid/hw07/

Download a copy of: hw07_caseid.tar file to the hw07 directory from http://blackboard.case.edu/ in the EECS337 homework assignment area. To untar the tar file type the command.
tar xvf hw07_caseid.tar

The following files will be created in the current working directory:
hw07_test.sh
math10.txt
math11.txt
math12.txt
math13.txt
math14.txt
math15.txt
math16.txt
math17.txt
math18.txt
math19.txt

Copy the following files from the assignment 06 directory to this directory with the commands:
cp ../hw06/Makefile .
cp ../hw06/first.l .
cp ../hw06/calc.c .
cp ../hw06/second.y .
cp ../hw06/yystype.h .

[Optional] If you did not complete the last assignment then download the hw06 solution from blackboard and use the source code in the hw06/ or hw06/extra/ directory. The source code in the extra subdirectory contains the extra credit (format) programming.

You are now ready to solve the laboratory assignment.

Part 3: Laboratory Assignment
Using your knowledge from the previous assignments, implement a symbol table and the assignment operations. If you are unsure how to proceed then follow these steps to complete the assignment.

Edit yystype.h and add to the YYSTYPE structure an integer index.

```
	int		index; // index into static symbol table
```

Below the YYSTYPE structure add a symbol table structure and extern the variables.

```
/*
 *	define a symbol table structure
 */
#define	SYMBOL_TABLE  struct symbol_table
SYMBOL_TABLE
{
#define	MAX_BUFFER_SIZE	128
	char	buffer[ MAX_BUFFER_SIZE];
	int	length;
	YYSTYPE	yylval;
};

/*
 *	extern variables
 */
extern SYMBOL_TABLE symbol_tables[];
extern unsigned int symbol_table_index;
```

Save the yystype.h file.

Edit first.l and add below the include file lines the symbol table declarations and identifier function. Complete the function to either find the identifier or create a new entry and return the index. (Hint: look at static/attribute.c from assignment 04). On create be sure to copy in the string, set the length field and to **set the symbol table YYSTYPE yylval.type field to IDENTIFIER and the yylval.index to the current symbol table index value**. You need these fields in order to print an identifier, which has not been assigned yet. Look at the yystype.h file for the YYSTYPE structure.

```
/*
 *	define the scanner symbol table (static)
 */
#define	MAX_IDENTIFIERS	128
SYMBOL_TABLE symbol_tables[ MAX_IDENTIFIERS];
unsigned int  symbol_table_index = 0;
/*
 *	scanner symbol table register function
 *	return an index into the symbol table (static)
 */
int	identifier( char *buffer, unsigned int length)
{
/*
 *	find the same string and return the index
 */
/*
 *	else update to the next symbol table field
 */
	return symbol_table_index;
};
```

Then add the identifier regular expression and action to the scanner.

```
{L}({L}|{D})*		{ yylval.type = IDENTIFIER; yylval.index = identifier(
yytext, yyleng + 1); return( IDENTIFIER); }
```

Save the first.l file.

Edit second.y and add the IDENTIFIER token to the list of tokens.

```
%token IDENTIFIER
```

Inside the first production action add the IDENTIFIER case to handle printing an identifier that has not been assigned yet.

```
              IDENTIFIER:
                     printf( "%s\n", symbol_tables[ $2.index].buffer);
                     break;
```

Below the first production, add the assignment operator and action.

```
       | lines ident '=' expr '\n'
       {
              symbol_tables[ $2.index].yylval = $4;
       }
```

Inside the expression production (expr) add the identifier and action below the `number` choice and before the semi-colon (`;`).

```
       | ident
       {
              $$ = symbol_tables[ $1.index].yylval;
       }
```

At the end of the productions add the identifier (ident) production for the IDENTIFIER token.

```
ident  : IDENTIFIER
       ;
```

Save the second.y file. (Notice the minimal changes to the parser code are designed based on the implementation of the symbol table.)

Test your calculator program using each of the test files by typing the commands:

make clean
make
./calc < ../hw06/math10.calc
./calc < ../hw06/math11.calc
./calc < ../hw06/math12.calc
./calc < ../hw06/math13.calc
./calc < ../hw06/math14.calc
./calc < ../hw06/math15.calc
./calc < ../hw06/math16.calc
./calc < ../hw06/math17.calc
./calc < ../hw06/math18.calc
./calc < ../hw06/math19.calc

You should get the same results as in the previous assignment.

Part 4: Extra Credit (5 points)
Edit the files again and change the static memory model for the symbol table into a dynamic memory model. Put most of your changes into the first.l file and extern any functions inside the yystype.h file.

Generate your output the same as in Part 5 and upload **ONLY** your first.l file to blackboard in the assignment area. Submit it with the comments **hw07** 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. Mark your output file with **EXTRA CREDIT** and turn in with your assignment.

Part 5: Laboratory Output Generation
When all your lab assignments have been completed execute the homework script file "./hw07_test.sh". To redirect the standard error (stderr) and standard output (stdout) to a file use the following command.
./hw07_test.sh &> hw07_test.txt

Print out the **hw07_test.txt** file and put your name, assignment number, date on it and turn it in with your homework exercises.

Your final directory structure for the calc compiler should be as below (using your Case ID):
EECS337/caseid/hw07/Makefile
EECS337/caseid/hw07/calc
EECS337/caseid/hw07/calc.c
EECS337/caseid/hw07/first.l
EECS337/caseid/hw07/hw07_caseid.tar
EECS337/caseid/hw07/hw07_test.sh
EECS337/caseid/hw07/hw07_test.txt
EECS337/caseid/hw07/math10.txt
EECS337/caseid/hw07/math11.txt
EECS337/caseid/hw07/math12.txt
EECS337/caseid/hw07/math13.txt
EECS337/caseid/hw07/math14.txt
EECS337/caseid/hw07/math15.txt
EECS337/caseid/hw07/math16.txt
EECS337/caseid/hw07/math17.txt
EECS337/caseid/hw07/math18.txt
EECS337/caseid/hw07/math19.txt
EECS337/caseid/hw07/second.y
EECS337/caseid/hw07/yystype.h