CS39003 Compilers Laboratory, Autumn 2024–2025
Assignment No: 3
Date: 02-Sep-2024

Parsing by YACC

In this assignment, you again work with lists. An input file (also called a program) consists of a sequence of lists. Each list is either an assignment statement that uses the keyword **set**, or a standalone arithmetic expression. A set statement looks like

(set lvalue rvalue)

where lvalue is a variable name, and rvalue is either a numeric constant or a variable name or an arithmetic expression. Assume that each numeric value is a signed integer. Variables use the same naming conventions as in C. The grammar for a program is given below. Here, PROGRAM is the start symbol. Each terminal symbol is colored red.

PROGRAM	⟶	STMT PROGRAM | STMT
STMT	⟶	SETSTMT | EXPRSTMT
SETSTMT	⟶	**(** **set** ID NUM **)** | **(** **set** ID ID **)** | **(** **set** ID EXPR **)**
EXPRSTMT	⟶	EXPR
EXPR	⟶	**(** OP ARG ARG **)**
OP	⟶	**+** | **–** | ***** | **/** | **%** | ******
ARG	⟶	**ID** | **NUM** | EXPR

All operators are assumed to be binary and to work on integer operands. The new operator ** is the exponentiation operator (write your own function for this operator).

Part 1: LEX program

Write a lex program expr.l to return the input tokens and to discard invalid characters in the input. Your lex program must return only the tokens for the terminal symbols. Here, you need to handle the keyword **set**. Use separate tokens for different binary operators (instead of a single token like OP).

Part 2: YACC program

Write a parser input expr.y to set up the tokens and data types and to specify the grammar. Actions involve handling the symbol table and creating and evaluating the expression tree. We detail these actions now.

Maintain a *single* symbol table for all ID's and NUM's that lex discovers in the input. Write functions to (i) add symbols to the table, (ii) set values to ID-type symbol-table entries, and (iii) read values of ID's and NUM's from the symbol table. You must design your own data structure for the symbol table (STL data types are not permitted).

As in Assignment 2, an expression tree is a binary tree. Each leaf node is of type ID or NUM, and should store a reference to the appropriate symbol-table entry. Each internal node is of type OP, and should store the exact operator (+ or – or ⋯) that the node stands for. You need to write functions to (i) create a leaf node (both child pointers should be NULL), (ii) create an internal (OP) node with the child pointers pointing to appropriate subtrees, and (iii) evaluate an expression tree. The parser generated by yacc would return pointers to appropriate nodes during the evaluation of an expression. Again, you must use your own data structure for the expression tree (STL data types are not permitted).

For a set statement, where the rvalue is a NUM or an ID, direct assignment is effected in the symbol-table entry for the lvalue. On the other hand, if the rvalue is an expression, an expression tree is build for it, and finally an evaluation of the expression tree will set the lvalue in the appropriate symbol-table entry. After the assignment, the expression tree will not be used again, and can be deleted.

For a standalone expression, an expression tree is to be prepared. When the entire expression is read, the tree is evaluated, the value is printed, and the tree can then be deleted.

Part 3: Your C/C++ program

Write the functions for manipulating the symbol table and expression trees in your C/C++ program with main(). In the yacc program, just declare the prototypes in order to make the actions meaningful to yacc. Also call yyparse() to run the input program.

Part 4: Makefile

Write a makefile with the targets all, run, and clean with the usual meanings. If you face a problem with make's default rule for lex files, just write the build commands for all one after another without using any dependency.

Submit a single zip/tar/tgz archive consisting of the four files written for the four parts described above. Do not submit the lex and yacc outputs or your a.out. Run make clean before preparing the archive.

Sample output

Input file (program)	Output of your C/C++ code

(set a 5)
(set b a)
(set c (+ (* a b) (+ a b)))
(+ (* a a)
  (** a b)
)
(set a 2)
(set b 9)
(/ (+ (** a 2) (** b 2))
  (+ (* a b) -1)
)
(set tmp b)
(set b (- (* 5 b) a))
(set a tmp)
(/ (+ (** a 2) (** b 2))
  (+ (* a b) -1)
)
(set tmp b)
(set b (- (* 5 b) a))
(set a tmp)
(/ (+ (** a 2) (** b 2))
  (+ (* a b) -1)
)
(set d (* (+ a b) (- a b) c))
	$ make all
	…
	$ make run
	…
	./a.out < sample.txt
	Variable a is set to 5
	Variable b is set to 5
	Variable c is set to 35
	Standalone expression evaluates to 3150
	Variable a is set to 2
	Variable b is set to 9
	Standalone expression evaluates to 5
	Variable tmp is set to 9
	Variable b is set to 43
	Variable a is set to 9
	Standalone expression evaluates to 5
	Variable tmp is set to 43
	Variable b is set to 206
	Variable a is set to 43
	Standalone expression evaluates to 5
	Error: syntax error
	$