Compiler Construction assignment

The task was to prepare a lexer parser for a 2048 like game.

The classic 2048 implementation involves only add operations. Tiles slide as per the players input and tiles with the same value combine to give double the value. Random tiles are generated if the move is valid (ie, the game state changes)

This can be broken down into elementary operations in multiple ways which conform to the original rules of the game. However, adding in the additional function of subtract causes tiles to be obliterated leading to different interpretations producing different results. Following are the interpretations implemented here.

1. SUBTRACT LEFT on 4224 giving 4400
    a. The tiles are slid to whatever extremity possible (ADD LEFT on [2][0][2][2] gives [2][2][2][0] here)
    b. The tiles are combined ([2][2][0][0] -> [4][0][2][0])
    c. The tiles are slid to extremities again ([4][0][2][0] -> [4][2][0][0])
2. SUBTRACT LEFT on 4224 giving 4040
    a. The tiles are slid to whatever extremity possible and are considered to be stacked atop each other should they be equal (ADD LEFT on [2][0][2][2] gives [22][2][0][0] here)
    b. The tiles are combined in one go ([22][2][0][0] -> [4][2][0][0])

Interpretation 2 considers that the tiles are not obliterated in the move, i.e. because the second sliding is not considered to be independent of combination, subtract left on 4224 results in the following operations. [4][2][2][4] -> [4][2 2][4][0] -> [4][0][4][0]

The code is split into 4 files. A scanner, a parser, a driver and a header file (which shall be included in all the 3 other files as it serves as a common interface between the 3). A make file has also been provided.

C++ was chosen as the final language to implement this assignment as opposed to C because of the additional functionality that it provides by way of the STL.

STEPS TO RUN THE PROGRAM:

1. On the terminal type : make cc
2. Then run the following command: ./cc 2>output.txt 1

The explanation for each of the arguments is provided on the last page.

THE LEXER:

The scanner is written in flex and the output file is a C++ file. The scanner tokenizes the keywords, punctuators, variables and numbers using the regex rules. Variable naming rules are the same as any other conventional programming language. All alphabets and underscore allowed, a number cannot start a variable name. Following this, we return a value corresponding to each token which is defined in a header

file (this header file parser.tab.hh is generated when generating the parser output and that acts as the interface between the lexer and the parser). In the case of numbers and variables, it won't be sufficient to return the type of token alone. It would also be necessary to return the value of that token. This is where the variable yylval comes in which takes the value of yytext (the scanned token). We also define YYTYPE as char*. YYTYPE is the data type of yylval. This is so that we can handle both variables and numbers (using atoi).

THE PARSER:

The parser is written in bison. The output of this includes a header file and a C++ file. The rules provided in the problem statement are implemented completely with a few additional specifications. Following are the production rules for the cfg from the parser along with a description of what they do (all caps represent terminals and the first letter capitalized represents non terminals):

1. Expression -> Operation Move END: This is for all possible moves. Add left, Add right, Subtract down and the remaining 13. This calls an extern move function with the requisite parameters. The move function internally handles the operations and the directions.
2. Expression -> ASSIGN Num TO Num COMMA Num END: This is for assigning a value to a tile. This handles an index out of bounds error case (when the numbers provided are not in the range [1,4]). There is also another case that needs to be handled. If a tile is assigned a value 0, this means that the tile is being obliterated. As a result, any variables holding that tile should be freed. This case is handled appropriately
3. Expression ->VAR VARIABLE IS Num COMMA Num END: This is used when a variable is to be allocated in memory. The existence of variables demanded the introduction of a symbol table. This is done by way of using a map from the C++ STL. The variable name is used as the key to index into the tile coordinates which is stored as a pair. The coordinates associated with the variable are expected to change as the tile moves because the tile is seen as a physical mobile entity. The tracking is taken care of the move functions which are implemented in the driver. The map is a shared extern variable and therefore has the flexibility of being accessed from multiple files.
4. Expression-> VALUE IN Num COMMA Num END: This is used to display the value of the tile at the requisite coordinate on the terminal. As was the case with ASSIGN, we handle the index out of bounds exception accordingly
5. Expression -> VARIABLE END: This rule was provided in the original specifications but was implemented to provide an easy interface to lookup any variable from the symbol table at any given time. If the variable exists, the tile value along with the coordinates are displayed. If the variable does not exist, we are shown the same accordingly.
6. Num->NUMBER: This is the case that where we directly pass a number in the above production rules
7. Num -> VALUE IN Num COMMA Num: This is specifically for handling nested queries (such as ASSIGN VALUE IN 1,1 TO 1,2.). This sets the result of the rule to be the value in the tile coordinate which can be used further in any other production. Naturally, we handle the index out of bounds exception here.
8. Operation ->ADD|SUBTRACT|MULTIPLY|DIVIDE: The 4 operations that were expected to be implemented. Add results in doubling, subtraction results in obliteration, multiplication results in squaring and division results in reducing the tile values to 1. Such combinations are only expected to occur if the tile values match up. The interpretations of how this is expected to happen has

been discussed above. This production results is a number which represents the expected operation to be passed in the move function. When we subtract, the possibility of obliteration arises. Should obliteration of a named tile occur, the variable corresponding to that tile is expected to be freed and that case is handled appropriately.

9. Move -> LEFT | RIGHT| UP| DOWN: The 4 move directions which are possible. This production results in a number which represents the expected direction to be passed on the move function.
10. EXIT END: This production rule is used to terminate the program. Pressing Ctrl+D marks the end of the input file and works just as well.

Additionally, the pre-defined error token was used in order to let the parser continue further processing after a syntax error had occurred. The expected stderr output is also written to stderr for every operation accordingly.

THE DRIVER:

The driver contains all of the function definitions and extern variable definitions (the board and the map). The board is implemented using a simple 2D 4x4 array for ease of access across the different files. Random number generation is and all the move functions are defined here. Move left, move right, move up and move down take in arguments to decide which operation must be performed. A function move determines which of these 4 functions must be called. This is the function that is called when the Operation Move END production rule is matched from the parser. An stderror_out function is also provided which just writes the output expected in stderror after every operation. A printBoard function is also provided to print the current state of the board. Further, we have a global variable mode which is responsible for deciding which interpretation we want the game to follow. Mode 1 follows subtract left on 4224 gives 4400. Mode 2 follows subtract left on 4224 gives 4040. Invalid argument counts and incorrect arguments are handled appropriately. Random number generated is seeded to system time. After generating the first random tile, the game begins and the function yyparse is called. This function is responsible for the entire meat of the work done by the program. It internally calls the lexer as well. As mentioned above, exit conditions are provided. Internal implementational details of the move functions and random tile generation functions are provided in the form of comments in the code

THE HEADER FILE:

The header file contains only those functions and variables which are referenced across all the files. Particularly, it has the printBoard, move and stderr_op functions as they are referenced in the parser and defined in the main. Same goes for the extern variables a[4][4] which represents the board state and the map <string,pair<int,int>> m which represents the symbol table. Standard header files are also included as a part of this header file so that they won't have to be reincluded in the parser, lexer or the driver.

THE MAKEFILE:

The make file consists of the commands needed to run to compile and run the code. Please run the following command on the terminal to make generate the executable.

make cc

Once the executable is generated, the following command can be run on the terminal to run the program:

./cc 2>output.txt 1

Here, the 2>output.txt part redirects the standard error output to a file output.txt. The 1 afterward is the command line argument which specifies which interpretation of the game you want to use. (this can be 2 as well).

Praveen Sridhar

2018A7PS0166G