Assignment 9

In the Simplesim assignment, we introduced a fictional machine called the Simplesim and the Simplesim Machine Language (SML). In this assignment, you are to write a compiler that converts programs written in a high-level programming language to SML. This will require you to use the techniques that you learned to convert infix expressions to postfix and to evaluate those postfix expressions. We provide you with programs in this new high-level language with the intention of compiling them on the compiler that you will write and running the resulting SML program on the simulator you wrote.

1. The Simple Language

Before we begin building the compiler, we discuss a simple, yet powerful, high-level programming language called Simple.

- Every Simple statement consists of a line number and a Simple instruction. Line numbers must appear in ascending order.
- Each instruction begins with one of the following Simple commands: `rem`, `input`, `data`, `let`, `print`, `goto`, `if...goto`, or `end`.
- Simple evaluates only integer expressions using the arithmetic operators `+`, `-`, `*`, and `/`. The operators have the same precedence as in C++. Parentheses can be used to change the order of evaluation of an expression.
- All variables in a Simple program are lower case single letters. Simple uses only integer variables and constants. Simple does not have variable declarations - merely mentioning a variable name in a program causes the variable to be declared and initialized to zero automatically.
- Simple uses the conditional `if...goto` statement and the unconditional `goto` statement to alter the flow of control during program execution. If the condition in the `if...goto` statement is true, control is transferred to the specified line of the program. The following relational operators are valid in an `if...goto` statement: <, >, <=, >=, ==, or `!=`. The precedence of these operators is the same as in C++.

Consider the following Simple program that accepts two input values and prints the maximum of the two:

```
10  rem
11  rem    print the maximum of two numbers
12  rem
20  data 10
30  rem
31  rem    get values
32  rem
40  input x
50  input y
60  rem
61  rem check x > y
62  rem
70  if x > y goto 111
80  rem
81  rem    y is maximum, print y
82  rem
90  print y
100 goto 130
110 rem
111 rem    x is maximum, print x
112 rem
120 print x
130 data 20
900 end
```

Each line in a Simple program must start with a line number. The line numbers must appear in ascending order and must always start in the leftmost column, i.e., no leading whitespace. Execution of a Simple program starts at the first line. In our example program above, this is line `10`, a `rem` command. `rem` commands are used to document the program and are not executed. Anything appearing after the word `rem` is simply ignored. Execution continues through the other documentation lines `11` and `12` and we reach line `20`, a `data` command. `data` commands, like `rem` commands are not executed. However, unlike `rem` commands, `data` commands do serve a purpose (explained below). `data` commands always end with a single integer. Execution proceeds through lines `20` and `30-32` and we approach line `40`, an `input` command.

`input` commands acquire a value from a `data` command and store that value in its variable. The very first `input` command that is executed uses the very first `data` command in the program. Subsequently executed `input` commands use the remaining `data` commands in the order in which they appear in the program. `data` commands are never used more than once. The same `input` command will consume many `data` commands if it is executed many times (e.g., in a "loop").

Line `40` is the first `input` command to be executed, so it acquires its value from the first `data` command (line `20`) and stores the value `10` in the variable `x`. Line `50`, our second `input` command to be executed, acquires its value from the next `data` command (line `130`) and stores the value `20` in the variable `y`. Execution continues through the documentation lines `60-62` and we approach line `70`, an `if...goto` command.

`if...goto` commands are always of the form `if <lop> <relop> <rop> goto <linenum>` where `<lop>` and `<rop>` are always either a single variable or constant (i.e., not expressions), `<relop>` is one of the relational operators listed above, and `<linenum>` is a line number appearing in the program.

Line `70` tests `x > y`. In our example program, this is false, so execution continues to line `80`. Had the test been true, then execution would have continued at line `111`. Note that line `111` is a `rem` command. Simple programs may branch (`if...goto` or `goto`) to `rem` and/or `data` lines, despite the fact that these lines are not executed. Execution continues through lines `80-82` and we approach line `90`, a `print` command. `print` commands print either a single variable or constant. In our example program, the number `20` is printed and execution continues to line `100`, a `goto` command, which branches (unconditionally) to line `130`, our second `data` command. Since `data` commands are not executed, we proceed to line `900`, an `end` command where program execution is terminated.

As a final example, consider the following Simple program that accepts a non-negative integer `x` and computes/prints the sum of integers from 1 through `x`.

```
10 rem sum 1 to x
15 input x
20 data 10
25 if x <= -1 goto 60
30 rem add x to total
35 let t = t + x
40 let x = x - 1
45 rem loop x
50 goto 25
55 rem output result
60 print t
99 end
```

As always, execution starts with first line of the program. The first executable statement is line `15`, an `input` command, which stores the value `10` (from the first `data` command on line `20`) in variable `x`. Execution continues with line `25`, an `if...goto` command where we test `x <= -1`. In our program this test is false, so execution continues to lines `35` and `40` which add `x` to `t` and decrement `x`, respectively. `t` is the variable that we are using to accumulate the sum. The variable `t` has not been explicitly initialized. Recall that variables that appear in a Simple program are initialized (implicitly) to 0. Line `50` unconditionally branches back to line `25` where we test `x <= -1`. Eventually, that test will become true, in which case control will be transferred to line `60`, the value in `t` (the sum) will be printed, and the program will terminate.

Simple does not provide a repetition structure (such as C++'s, `for` or `while`). However, Simple can simulate each of C++'s repetition structures using the `if...goto` and `goto` commands as we did in the example above.

2. Input

The input to your program is a Simple program, just like the ones shown in the previous section. You may assume that all Simple programs given to your compiler are syntactically correct. Syntax checking is a critical component of any compiler, however, proper syntax checking is a very difficult problem and well beyond the scope of this course. Hence, when you write your compiler you may make the simplifying assumption that the programs you are compiling are syntactically correct. You may also assume

1. all lines are numbered uniquely and appear in ascending order based on that line number,
2. all line numbers start in the leftmost column,
3. all line numbers appearing in `if...goto` and `goto` commands are in the program,
4. there is at least one space between a line number and the command,
5. there is at least one space between the command and the rest of the line,
6. for `let` commands, there is at least one space before and after the = sign (note there may or may not be spaces in the expression on the right of the = sign),
7. and that for `if...goto` commands, there will be at least one space separating the left and right operands from the relational operator.

3. Output

Your compiler will produce a complete SML program, including the `-99999` line followed by any input (from any data in the Simple program) or an error message (error messages described in **Section 4**). The SML program generated by your compiler must be immediately suitable for execution on your Simplesim. In other words, you should be able to take the SML program generated by your compiler and "feed it" directly (without modification) as input to your `simplesim` program from Assignment 4.

4. The Two-Pass Compiler

The result of the compiler is the SML program, which is composed of SML instructions and data, a line containing `-99999` to mark the end of the program, and possibly some input data. Your compiler program should use a memory array identical to the one it used in the Simplesim assignment and a data array with a counter to collect the values from the Simple `data` commands.

For example:

```
int memory[MEMSIZE];
int data[MEMSIZE];
int ndata;
```

Your program will also use an array of flags and a symbol table (both are described below). The symbol table will have 1,000 "rows", where each row is an instance of the structure below. The flags array must have a flag for each memory location.

```
#define SYMBOL_TABLE_SIZE 1000

struct table_entry
{
    int symbol;
    char type;        // 'C' constant, 'L' Simple line number, 'V' variable
    int location;     // Simplesim address (00 to MEMSIZE-1)
};

table_entry symbol_table[SYMBOL_TABLE_SIZE];
int flags[MEMSIZE];
```

After some minor initialization, the compiler performs two passes. The first pass constructs a symbol table in which every *line number*, *variable name*, and *constant* of the Simple program is stored with its type and and corresponding location in the final SML code (the symbol table is described in detail below).

Constants and variables are placed at the *bottom* of the Simplesim's memory (memory locations 99, 98, 97, ...) and the SML program (instructions) are placed at the top of the Simplesim's memory (locations 00, 01, 02, ...). The first pass also produces the corresponding SML instruction(s) for each Simple statement. This results in a "hole" in the middle of the Simplesim's memory (between the last SML instructions and the constants/variables) that will be used as stack space to evaluate the arithmetic expressions in the `let` commands.

As we will see, if the Simple program contains `let` statements or statements that transfer control to a line later in the program, the first pass results in an SML program containing some partial instructions and it uses the `flags` array to mark those instructions.

Your program should use the following variables to record the Simplesim address for the next instruction, the next constant or variable, and an index for the next symbol in the symbol table.

```
int next_instruction_addr;
int next_const_or_var_addr;
int next_symbol_table_idx;
```

The second pass of the compiler locates (using the `flags` array) and completes (using the symbol table and stack space) the partial instructions. After the second pass, the compiler prints the finished SML program, followed by any input data.

4.1. Initialization

Your compiler must initialize all of the Simplesim's memory to 7777 and all the flags to -1. It should then initialize the next instruction address to the "top" of the Simplesim's memory (memory location 0), the next constant or variable address to the "bottom" of the Simplesim's memory (memory location `MEMSIZE-1`), the index of the next entry into the symbol table to 0, and finally the counter for the Simple data commands to 0.

For example:

```
next_instruction_addr = 0;
next_const_or_var_addr = MEMSIZE-1;
next_symbol_table_idx = 0;
ndata = 0;
```

4.2. Pass 1

The first pass of the compiler reads and processes the Simple program one line at a time. In processing *a single line* of the Simple program, the compiler can make addition(s) to the symbol table, add (possibly *partial*) SML instruction(s) to the Simplesim's memory, and add constants and/or variables to the Simplesim's memory.

The general structure of the first pass should look like something like this:

```
string buffer1, buffer2, line_number, command;

while (getline(cin, buffer1))
{
    buffer2 = buffer1;         // buffer2 used for 'let'
    istringstream ss(buffer1);

    ss >> line_number;

    // ... code to add line_number to symbol table, type 'L' ...

    ss >> command;

    if (command == "input")
    {
        // ... code to process 'input' command ...
    }
    else if (command == "data")
    {
        // ... code to process 'data' command ...
    }
    else if (command == "let")
    {
        // ... code to process 'let' command ...
    }
    else if (command == ...)
    {
        .
        .
        .
    }
    else if (command == "rem")
    {
        // ... code to process 'rem' command ...
    }
}
```

(The `istringstream` class is defined in the header file `<sstream>` and is part of the standard namespace.)

Each time a new line is read from the Simple program, its line number must be added to the symbol table along with the corresponding address of the next instruction of the SML program.

Each time a constant and/or variable is encountered, first the symbol table is checked to see if the constant or variable already exists in the symbol table.

- If the constant or variable does not exist within the symbol table (i.e., this is the first time that it has been encountered in the program) a Simplesim memory cell is allocated for it (initializing the memory to 0 if it was a variable) and the symbol table is updated by adding the variable or constant along with its memory location.
- If the variable or constant already exists in the symbol table, then no additional memory is allocated for it and the symbol table is left unmodified.

4.2.1. Processing Simple Commands

Here we discuss the details of how to process each of the Simple commands. In discussing each, it is assumed that the line number of the command has already been added to the symbol table.

- `rem`. There is nothing left to do after adding the line number to the symbol table.
- `data`. Add the integer appearing at the end of the line to the `data` array and increment `ndata`. If there is not enough room in the `data` array to add this element, print `*** ERROR: too many data lines ***` and terminate the compilation immediately, i.e., without printing the SML program.
- `input`. Search the symbol table for the variable appearing at the end of the line. If the variable appears in the symbol table, then extract the variable's Simplesim memory location from the table. If the variable did not appear in the symbol table, allocate a Simplesim memory cell for it at `next_const_or_var_addr` and add it to the symbol table. Be sure to initialize the Simplesim's memory for that variable to 0 and *decrement* `next_const_or_var_addr` for the next variable or constant. Be sure to note the newly-allocated memory location of the new variable (by saving it in the symbol table).

  Using the variable's address, whether it was extracted from the symbol table or it was newly allocated, place a `READ` instruction at memory location `next_instruction_addr`. Be sure to increment `next_instruction_addr` for the next SML instruction.
- `print`. Process this in exactly the same manner as the `input` command, using the address from the symbol table if the variable already existed there or creating space for it and adding to the symbol table. The only difference in processing a `print` command is that you should place a WRITE instruction in memory rather than a READ.
- `goto`. Search the symbol table for the line number appearing at the end of the line. If the line number appears in the symbol table (i.e., it refers to a line "before" this `goto` command), then extract the line number's memory location from the symbol table and place/generate a `BRANCH` instruction at memory location `next_instruction_addr`. If the line number was not in the symbol table, then it refers to a line "after" this `goto` command. This is called a *forward reference*. In this case, we place/generate a partial `BRANCH` instruction (omitting the address to branch to) and flag the instruction to indicate that the second pass of the compiler must complete the instruction. Use the `flags` array and set `flags[next_instruction_addr]` to the line number this `goto` command is supposed to branch to. In either case, after writing the `BRANCH` instruction (partial or otherwise) be sure to increment `next_instruction_addr` for the next SML instruction.
- `if...goto`. Compilation of the `if...goto` and `let` statements is more complicated than the other statements - they are the only statements that produce more than one SML instruction. For an `if...goto` statement, the constants and/or variables appearing in the test are searched for in the symbol table and added to the Simplesim's memory and symbol table if necessary (noting the Simplesim memory locations of each). Then the compiler produces SML instructions (typically a `LOAD` followed by a `SUBTRACT`) to test the condition and branch if appropriate. The result of the branch could be a forward reference and should be handled just like the forward references in the `goto` command, i.e., writing partial `BRANCHZERO` and/or `BRANCHNEG` instructions and flagging them with the line number in the `flags` array. Each of the relational operators can be simulated using SML's `BRANCHZERO` and `BRANCHNEG` instructions (or possibly a combination of both). Be sure to increment `next_instruction_addr` after each SML instruction.
- `let`. Compilation of the `let` statement is the most challenging. It not only produces more than one SML instruction but also the evaluation of the expression (`rhs`) requires you to manage your own stack (as an array) using the "hole" in the Simplesim's memory between the SML instructions and the constants/variables. The problem is further complicated by the fact that during the first pass we do not know where that stack space starts, so we are forced to write *partial* SML instructions and flag them using the `flags` array.

  The first thing you must do in processing a `let` statement is to search the symbol table for the variable being assigned. If it is not in the symbol table, then a Simplesim word must be allocated for it, initialized to 0, and it should be added to the symbol table. Then, using the copy of the statement (`buffer2`), call your `convert()` function from the previous assignment, passing it the address of the first non-whitespace character *after* the equal sign, to get a postfix expression. Evaluate the postfix expression using the stack algorithm to write the SML instructions (described in detail below).

  The actual Simplesim memory location that will be the start of our stack cannot be known until after all the space has been allocated for constants and variables (i.e., after the first pass has been completed). Once that is done, then we can use the "hole" between the the last SML instruction and the last constant/variable. The stack will start above (but not *immediately* above) the last constant/variable added and grow "upward" toward the SML instructions each time we "push" a value. The first Simplesim word immediately above the last constant/variable is specially reserved for the non-commutative operators `-` and `/` (explained below), so the stack actually starts immediately above that specially reserved word.

  Start the processing of the postfix expression by initializing a stack index integer to 0 (`next_stack_idx = 0`), which represents the index into the stack where the next pushed value should go. Now you are ready to start processing the postfix expression.

  Processing Operands

  Each time you encounter an operand (constant or variable), you must search the symbol table for it, and if it is not there, you must add it to the Simplesim's memory and the symbol table. Extract the operand's memory location from the symbol table.

  Operands are pushed onto the stack. Since there are no memory-to-memory SML instructions, pushing a value onto the stack requires the use of the accumulator, i.e., you must `LOAD` and then `STORE`. Write the `LOAD` instruction using the operand's address from the symbol table, then write a *partial* `STORE` instruction (omitting the address). Use the `flags` array to flag this `STORE` instruction by setting

  ```
  flags[next_instruction_addr] = -3 - next_stack_idx;
  ```

  Recall that the `flags` array was initialized with `-1` and that it uses positive integers to represent forward referenced line numbers. This leaves negative numbers < `-1` to represent stack indices. When the `flags` array element has the value `-2`, that refers to the specially reserved word for processing non-commutative operators (explained below). Values < `-2` are used to represent stack indices, e.g., `-3` means `stack + 0`, `-4` means `stack + 1`, `-5` means `stack + 2`, etc.

  Thus, processing an operand requires finding its SML address from the symbol table (adding it if necessary), writing the `LOAD` instruction using that address, writing a partial `STORE` instruction, setting the `flags` array element corresponding to that `STORE` instruction to `-3 - next_stack_idx`, and incrementing `next_stack_idx`.

  Processing Operators

  Processing operators requires you to pop the two operands, perform the operation, and push the result.

  Processing the commutative operators (`+` and `*`) is a little easier than the non-commutative operators (`-` and `/`). Commutative operators pop the stack, placing the value into the `accumulator` (`LOAD`), and then perform the operation by popping the stack a second time and applying the operation to the value in the `accumulator` (`ADD` or `MULTIPLY`) leaving the result in the accumlator. That result is pushed back onto the stack (`STORE`). Therefore, processing a commutative operator writes three *partial* SML instructions (omitting the address in all three).

  For example, the code for processing the operator `+` would look like this:

  ```
  memory[next_instruction_addr] = LOAD * 100;     // omit address
  next_stack_idx--;
  flags[next_instruction_addr] = -3 - next_stack_idx;
  next_instruction_addr++;

  memory[next_instruction_addr] = ADD * 100;      // for addition, omit address
  next_stack_idx--;
  flags[next_instruction_addr] = -3 - next_stack_idx;
  next_instruction_addr++;

  memory[next_instruction_addr] = STORE * 100;    // omit address
  flags[next_instruction_addr] = -3 - next_stack_idx;
  next_stack_idx++;
  next_instruction_addr++;
  ```

  (Code for processing the operator `*` is nearly identical - just change `ADD` to `MULTIPLY`.)

  Processing the non-commutative operators requires more care. Recall that the operand on the top of the stack is the *right* operand. Therefore, we cannot simply load that value into the `accumulator` and apply the operation using the next value on the stack. We must pop the right operand (`LOAD`) and temporarily store (`STORE`) it in the special memory location sitting just beneath the stack. Then pop the stack again, placing the left operand into the `accumulator` (`LOAD`). Apply the operator using the value in the special memory location (`SUBTRACT` or `DIVIDE`), leaving the result in the `accumlator`, and then push the result back onto the stack (`STORE`). Therefore, processing a non-commutative operator writes five *partial* SML instructions (omitting the address in all five).

  For example, the code for processing the operator `-` would look like this:

  ```
  memory[next_instruction_addr] = LOAD * 100;        // omit address
  next_stack_idx--;
  flags[next_instruction_addr] = -3 - next_stack_idx;
  next_instruction_addr++;

  memory[next_instruction_addr] = STORE * 100;       // omit address
  flags[next_instruction_addr] = -2;
  next_instruction_addr++;

  memory[next_instruction_addr] = LOAD * 100;        // omit address
  next_stack_idx--;
  flags[next_instruction_addr] = -3 - next_stack_idx;
  next_instruction_addr++;

  memory[next_instruction_addr] = SUBTRACT * 100;    // for subtraction, omit address
  flags[next_instruction_addr] = -2;
  next_instruction_addr++;

  memory[next_instruction_addr] = STORE * 100;       // omit address
  flags[next_instruction_addr] = -3 - next_stack_idx;
  next_stack_idx++;
  next_instruction_addr++;
  ```

  (Code for processing the operator `/` is nearly identical - just change `SUBTRACT` to `DIVIDE`.)

  Once you have finished evaluating the postfix expression. the answer is sitting on the top of the stack, where it must be removed and placed into the memory location of the variable of the `let` command. This is done with two SML instructions, a *partial* `LOAD` followed by a full `STORE`. The element of the `flags` array that corresponds to the partial `LOAD` instruction should be set to `-3` (i.e., the top of the stack).
- end. Simply write a `HALT` instruction at memory location `next_instruction_addr`. Be sure to increment `next_instruction_addr` after writing the instruction.

There are only 100 words in the Simplesim memory and it is possible to deplete that. If by adding instructions you find that you have run past the end of memory or have entered the variable/constant section of the memory, or if by allocating space for variables/constants you have entered the program section of the memory, simply print the message `*** ERROR: ran out Simplesim memory ***` and terminate the compilation immediately, i.e., without printing the SML program.

4.2.2. First Pass Example

To illustrate the first pass of the compiler, we take you through its steps (see **Figure 1** below) as it processes the summation example program from **Section 1**.

We start with the first line of the program, line 10. As is done with each line of the program, the line number is added to the symbol table (see **Figure 3**), and since it is a `rem` command, nothing else is done.

Line 15 is an `input` command. The symbol table is searched for the variable `x` and it is not found (at this point there are only two entries in the symbol table, lines 10 and 15). Since `x` was not in the symbol table, a Simplesim word is allocated for `x` (at location 99), it is initialized to 0, and `x` is added to the symbol table. We are now ready to write our first SML instruction, a `READ` command using the address of `x` from the symbol table. Since we were able to write a full SML instruction (i.e., complete with address) we do not flag the instruction, meaning we leave this element of the `flags` array unchanged.

Compilation continues with line 20, a `data` command. The value (10) is placed into the first element of the `data` array and `ndata` is incremented.

Proceeding to line 25, we reach an `if...goto` command. The symbol table is searched for the left operand `x`, which is found. The symbol table is then searched for the right operand, the constant `-1`. It is not found in the symbol table, so the next available Simplesim word is allocated for `-1` (at location 98), the word is initialized to -1, and the constant `-1` is added to the symbol table. We are now ready to start writing the SML instructions. This particular `if...goto` command is testing <=. That test is performed by subtracting the right operand from the left and branching if the difference is zero or negative (other `if...goto` tests are handled in different, yet similar ways). The addresses of the left and right operands (locations 99 and 98) are extracted from their symbol table entries and used in the `LOAD` and `SUBTRACT` instructions. Now to write the `BRANCHNEG` and `BRANCHZERO` instructions. The symbol table is searched for the specified line number (`60`). Since this is a forward reference (referring to a line "below" us), line 60 does not appear in the symbol table. Hence, we are forced to write partial branching instructions (omitting the address) and we must flag these instructions by placing the line number (60) into the cells of the `flags` array that correspond to the branch instructions' memory locations (`flags[03] = 60;`, `flags[04] = 60;`).

Continuing through line 30 (another `rem` command), we encounter our first `let` command in line 35. The symbol table is searched for the variable appearing on the left side of the equal sign (`t`) and it is not found. Simplesim memory location 97 is assigned to the variable `t`, it is initialized to 0, and `t` is added to the symbol table. We extract its Simplesim address (97) from the symbol table and save it for future use (the `STORE` operation at the end of processing this instruction). The infix expression `t + x` is converted to its postfix equivalent `t x +` and compilation continues by processing the postfix expression, one symbol at a time.

The first symbol (`t`) is an operand, so we must push it onto the stack. This requires generating a `LOAD` and a `STORE`. The symbol table is searched for the symbol `t` and it is found. Its Simplesim address is extracted from the symbol table (97) and used in the `LOAD` instruction. Now we must `STORE` the value in the stack. Since we do not know the Simplesim address where the stack will begin (we cannot know that until after the end of the first pass when space for all the variables/constants has been allocated), we are forced to write a *partial* `STORE` instruction (omitting the address) and flagging this instruction by placing a representation of the stack index (0) into the cell of the `flags` array corresponding to this instruction's memory location (`flags[06] = -3 - next_stack_idx;`) and the stack index is incremented. This concludes "pushing" the first operand (`t`) onto the stack and we continue with the next symbol in the postfix expression, `x`.

`x` is also an operand and must be pushed onto the stack. The symbol table is searched for the variable `x`. It too is found, and its Simplesim address (99) is extracted from the symbol table and is used in the `LOAD` instruction. Once again, we are forced to write a *partial* `STORE` instruction, this time using the next slot in our stack (stack index 1), of course incrementing the stack index after writing the *partial* `STORE` and flagging its corresponding cell of the `flags` array. This concludes pushing the second operand (`x`) onto the stack.

Processing of the postfix expression continues as we move on to the next symbol, the commutative binary operator `+`. Processing such an operator requires us to pop the stack twice (performing the addition on the second pop) and pushing the result back onto the stack. We perform the first pop by decrementing the stack index and writing the *partial* `LOAD` instruction. We perform the second by by decrementing the stack index and writing the *partial* `ADD` instruction. The sum is now sitting in the `accumulator` and must be pushed back onto the stack. This is performed by writing the *partial* `STORE` instruction and then incrementing the stack index.

This concludes the processing of the postfix expression. The stack should still have one value left, the value of the entire postfix expression. We must pop the stack and place the value into the variable that is the target of the `let` statement, in this case `t`. We pop the stack with a *partial* `LOAD` instruction and conclude with a full `STORE` instruction (using the address 97 that we extracted from the symbol table when we started processing this `let` command).

Compilation continues to line to 40 where we encounter our second `let` instruction, this time involving the non-commutative binary operator `-`. We start again by searching the symbol table for the variable appearing to the left of the = sign (`x`) and find it, noting its memory location (99). We continue by converting the infix expression `x - 1` to its postfix equivalent `x 1 -` and processing it one symbol at a time. The first two symbols of this postfix expression are also operands and they are both pushed onto the stack (SML instructions `14-17`) the same way we pushed the first two operands in the previous `let` statement, with one minor difference. The constant `1` is not found in the symbol table, so Simplesim memory location 96 is allocated for it, set to 1, and the constant is added to the symbol table.

Now we must process the non-commutative binary operator `-`. Recall that the value on the top of the stack is the *right* operand. We cannot simply load that value into the `accumulator` and perform the subtraction operation using the second value from the stack. We must pop the right operand from the stack with a *partial* `LOAD` instruction and store it into the special memory location sitting just below the stack with a *partial* `STORE` instruction (specifying the special memory location in the `flags` array, `flags[19] = -2;`). Processing continues by popping the next value from the stack into the `accumulator` with a *partial* `LOAD` instruction and then writing the *partial* `SUBTRACT` instruction using the special memory location where we temporarily stored the right operand (`flags[21] = -2;`). Finally, the difference is pushed back onto the stack with a *partial* `STORE` instruction. This concludes processing of the postfix expression and the result is popped off the stack and stored in `x` (SML instructions `23-24`) in the same manner as the previous `let` command.

We continue through line 45 to line 50 where we encounter our first `goto` command, which always results in a `BRANCH` instruction. The only thing to determine is whether or not we can write a full instruction. We search the symbol table for the specified line number (25) and it is found. This allows us to write a full `BRANCH` instruction using the address that we extract from the symbol table. If the line number was not in the symbol table (e.g., a forward reference), we would have been forced to write a *partial* `BRANCH` instruction (omitting the address) and flag the instruction, setting `flags[25]` to the forward referenced line number.

Continuing to line 60, a `print` command, we search the symbol table for the specified variable (`t`). In this case, the variable was found. However, if the variable was not in the symbol table, space would have to be allocated for it and the symbol table would be updated. In either case, `print` statements always result in a full `WRITE` instruction. We conclude the first pass with line 99, an end command which generates a `HALT` instruction.

This concludes the compiler's first pass through the summation program. At the end of the first pass, `data[0] = 10` and `ndata = 1` (from the `data` command in line 20), the symbol table appears as it is presented in **Figure 3**, and the Simplesim `memory` and `flags` arrays appear as presented in **Figure 2**. There we can see that the SML program will ultimately occupy Simplesim memory locations `00-27` and that the constants/variables occupy `96-99`, leaving a "hole" in the memory (`28-95`) between the two. This "hole" will be used for the stack (starting at memory `94` and growing up to and including memory location `28`) and the special temporary holding cell for right operands when evaluating non-commutative binary operators `-` and `/` (memory location `95`). Now that we have identified the boundaries of the "hole", we are ready to commence the second pass of our compiler.

Simple program	SML	flags
10 rem sum 1 to x		
15 input x	00 1199	
	99 0000	
20 data 10		
25 if x <= -1 goto 60	01 2299	
	02 3298	
	03 4300	60
	04 4200	60
	98 -0001	
30 rem add x to total		
35 let t = t + x	05 2297	
	06 2100	-3 (stack + 0)
	07 2299	
	08 2100	-4 (stack + 1)
	09 2200	-4 (stack + 1)
	10 3100	-3 (stack + 0)
	11 2100	-3 (stack + 0)
	12 2200	-3 (stack + 0)
	13 2197	
	97 0000	
40 let x = x - 1	14 2299	
	15 2100	-3 (stack+0)
	16 2296	
	17 2100	-4 (stack+1)
	18 2200	-4 (stack+1)
	19 2100	-2 (right operand)
	20 2200	-3 (stack+0)
	21 3200	-2 (right operand)
	22 2100	-3 (stack + 0)
	23 2200	-3 (stack + 0)
	24 2199	
	96 0001	
45 rem loop x		
50 goto 25	25 4101	
55 rem output result		
60 print t	26 1297	
99 end	27 4400	

Figure 1: First pass of summation program

Location	memory	flags
00	+1199	
01	+2299	
02	+3298	
03	+4300	60
04	+4200	60
05	+2297	
06	+2100	-3
07	+2299	
08	+2100	-4
09	+2200	-4
10	+3100	-3
11	+2100	-3
12	+2200	-3
13	+2197	
14	+2299	
15	+2100	-3
16	+2296	
17	+2100	-4
18	+2200	-4
19	+2100	-2
20	+2200	-3
21	+3200	-2
22	+2100	-3
23	+2200	-3
24	+2199	
25	+4101	
26	+1297	
27	+4400	
	"hole"	
96	+0001	
97	+0000	
98	-0001	
99	+0000	

Figure 2: `memory` and `flags` arrays after first pass of summation program. Values stored in `memory` array are `+7777` and values stored in `flags` array are `-1` unless otherwise noted.

Symbol	Type	SML Address
10	L	00
15	L	00
x	V	99
20	L	01
25	L	01
-1	C	98
30	L	05
35	L	05
t	V	97
40	L	14
1	C	96
45	L	25
50	L	25
55	L	26
60	L	26
99	L	27

Figure 3: Symbol table after first pass of summation program

4.3. Pass 2

At the end of the first pass, you will know the address of the last constant/variable that space was allocated for. The Simplesim word immediately before that (location `95` from the example in **Section 4.2**) becomes the location for the right operand of the non-commutative binary operators and the word immediately before that (location `94` from **Section 4.2**) is the starting location for the stack. You will need that information for the second pass.

The purpose of the second pass is to complete the partial instructions written in the first pass. This is done by traversing the `flags` array and completing any instruction whose corresponding `flags` value is `!= -1`.

A positive `flags[i]` value represents a line number that was a forward reference. Search the symbol table for the line number, extract its address, and complete the SML instruction (`memory[i]`) by adding the address.

A `flags[i]` value of `-2` represents the special right operand memory location. Complete the SML instruction (`memory[i]`) by adding the address of that word (location `95` from **Section 4.2**).

When `flags[i] < -2`, that represents a memory location in the stack. Calculate the stack index `idx = -3 - flags[i]` and use that index to compute the address. Using the example from **Section 4.2**, `flags[i] = -3` produces a stack index of 0, which corresponds to memory location `94`, `flags[i] = -4` produces stack index 1 and that corresponds to memory location `93`, and so on. Complete the SML instruction (`memory[i]`) by adding the calculated stack address.

There is one error that could possibly be detected during the second pass: the "hole" may not be big enough to accommodate the stack space necessary to evaluate the postfix expressions. In that case, your program should print `*** ERROR: insufficient stack space ***` and terminate the compilation immediately, i.e., without printing the SML program.

4.4. Print SML Program

After the second pass you will have a complete SML program (no *partial* instructions) and a `data` array ready for printing. Print the *entire* `memory` array (one word per line), followed by `-99999`, and conclude by printing all the values read into the `data` array.

5. Files We Give You

By running the `setup` command for this assignment, you will receive a makefile, a renamed version of the main routine for Assignment 4 (`simplesim_main.cpp`) to be used in building your `simplesim` executable, and a collection of Simple programs. They include six Simple programs that will compile (including the two example Simple programs `sum.s` and `max.s`) and a collection of Simple programs that will not compile. The file name and brief description of each Simple program is in the table below.

Filename	Expected Result	Description
Simple programs that will not compile:		
`bigdata.s`	`*** ERROR: too many data lines ***`	Simple program has too many `data` lines.
`bigpgm.s`	`*** ERROR: ran out Simplesim memory ***`	Cannot allocate space for an instruction.
`bigpgmvar.s`	`*** ERROR: ran out Simplesim memory ***`	Cannot allocate space for a variable.
`bigpgmcmd.s`	`*** ERROR: ran out Simplesim memory ***`	Cannot allocate space for an instruction.
`bigpgmstack.s`	`*** ERROR: insufficient stack space ***`	Insufficient stack space for single `let` statement.
Simple programs that compile:		
`end.s`	no Simple output	One line Simple program, end.
`read.s`	print `READ`: value	Reads an integer into a variable.
`rw.s`	`READ` and output value then constant	Reads and prints a number and constant.
`max.s`	max of two input values	Reads two numbers and prints max.
`sum.s`	sum of from 1 to x	Reads number (x) prints sum from 1 to x.
`prime.s`	1 if x is prime, else -1	Prints 1 if input is prime, otherwise -1.

You will receive two executable files, `simplesim_check`, an executable Simplesim, and `scc_check`, an executable compiler solution to this assignment. You may use the Simple programs above along with the solutions `simplesim_check` and `scc_check` to debug your program. The output of your compiler and Simplesim must look exactly like the output produced by `scc_check` and `simplesim_check`. Your programs will be tested in the following manner,

```
z123456@turing:~$ ./scc_check < sum.s | ./simplesim_check > sum.key
z123456@turing:~$ ./scc < sum.s | ./simplesim > sum.out
z123456@turing:~$ diff sum.out sum.key
z123456@turing:~$
```

where `scc` and `simplesim` are your solutions. To be eligible for full credit your output must exactly match (empty `diff`) the output of `simplesim_check` for each of the Simple programs listed in the table above. Note this is a *minimum* requirement. We may test your programs with Simple programs other than those listed above.

6. Files You Must Write

You must place `simplesim.cpp`, `simplesim.h`, `sml.h` (from Assignment 4), `inpost.cpp`, `inpost.h`, `mystack.cpp`, and `mystack.h` (from Assignment 7) in your submission directory for this assignment. You will also submit a new file for this assignment, `scc.cpp` which must `#include "sml.h"` and use the `#define` values for each of the Simplesim instructions (e.g., `LOAD`, `STORE`, ...), and it must `#include "inpost.h"` and call the `convert()` function in `inpost.cpp`. We will re-make your `simplesim` executable and your `scc` compiler using the makefile that we supply you.

Note that your program will be tested using your solutions to the previous assignments. You will need to turn in working versions of them in order to receive full credit for this assignment (note that you must supply them again when turning in this assignment - if they don't work, then you must fix them).

7. Hints

The structure of this program is left largely up to you. If you want to write it as a class in a fashion similar to what was done on Assignment 4, you are welcome to modify the makefile to make that possible, or you can put both the class definition and the member function definitions for your class in `scc.cpp`. You are equally welcome to write one giant `main()` routine in `scc.cpp`. In practice, you're probably going to want to write at least a few other functions besides `main()` to avoid massive amounts of duplicated code.

The most complicated part about this assignment is the first pass of the compiler. The initialization, second pass, and outputting the program are all relatively straightforward. You might want to implement those three sections right away and start with just the `while (gets(cin, buffer1))` loop of the first pass (described in **Section 4.2**), leaving the body of the loop initially empty. Continue by adding the code to process Simple commands, one at a time, to the body of the loop. Always convince yourself that what you have done to that point is working correctly before proceeding. Below is a suggested strategy you might take using the Simple programs we have provided.

1. Start with `rem` and `end`. This will get you to register each line number in the symbol table. Verify with `end.s` and `bigpgm.s` before proceeding.
2. Add `input` and `data`. You will also have to write the code to handle variables as well. Verify with `read.s`, `bigdata.s`, `bigpgmvar.s`, and `bigpgmcmd.s` before proceeding.
3. Add `print` and `goto`. Verify with `rw.s` before proceeding. `rw.s` requires you to process a constant and forward reference. These are the last Simple commands that produce only single SML instructions.
4. Add `if...goto`. You might want to implment even this single Simple command in stages, adding one relational operator at a time. Verify with `max.s` before proceeding. This will require you to at least implement the > relational operator.
5. Conclude with `let`. This is the most complicated Simple command to compile. Start by verifying with `sum.s` and `bigpgmstack.s`. This will require you to implement the arithmetic operators + and - and the <= relational operator. After convincing yourself that your compiler works with both of those Simple programs, finish up with `prime.s` which introduces the two arithmetic operators `*` and `/` and the relational operators ==, <, and >.
6. Verify that your code for the other relational operators works using the remaining test programs.

The Simple programs that we have provided are there to help you. They are not intended to be a complete test suite for your program. You should write your own Simple programs to further test your compiler. When grading your program we may use your compiler to compile programs other than the ones that we supplied.

There is one other warning we should issue. When searching the symbol table for a particular symbol, be sure that you check both the symbol *and* its type for a match. The ASCII value for the variable `'a'` is 97, `'b'` is 98, and so on. Those values can easily be mistaken for line numbers. Also, searching for a constant 10 could be mistaken for line number 10.