CS 3323 - Principles of Programming Languages
Assignment 4

The objective of this programming assignment is to add basic control-flow functionality to a simple compiler being developed in this course. Instructions are provided below.

First, download the files grammar.y, scanner.yy, icode.cc, icode.hh, symtab.cc, symtab.hh, inputs-outputs.tar.gz, Makefile, run-all.sh and driver.cc from the assignment directory space. Then, perform the necessary modifications to grammar.y to add support for the repeat-until, while-do and if-then-else programming constructs. You should only work on the grammar file (grammar.y).

The assignment is due on April 29th, 2020, 11:59pm. Late policy deduction applies.

Perform the following modifications to the compiler:

1. (0.5pt) This version of the compiler introduces 5 new intermediate code operations: OP_LT, OP_GT, OP_JMP, OP_JNZ and OP_JZ. See the file icode.cc, lines 296–354. Read their implementation to be sure you understand their semantics. The corresponding constant definitions and instruction codes have been added to icode.hh, and can be used as new intermediate code operations.

   Also observe the new rule l_expr in grammar.y, which allows to compare two integer expressions and store the result of this comparison in an intermediate variable. The action stores a pointer to the intermediate variable generated for any other rule to use it (e.g. repeat-until).

   A few small modifications have been performed to the loop executing the intermediate operations, and they mainly pertain to the execution of conditional and unconditional jumps: changing the program counter (now explicitly **pc**) by some instruction position given by an argument of the intermediate code (always the 3rd argument).

2. (1.5pt) In grammar.y, complete the semantic actions for the repeat-until construct (see construct_repeat rule). The construct **jumps backwards** to the first instruction of the loop's body whenever the condition is false. This means that you must use the new intermediate code operation OP_JZ (jump if zero). As the jump target is to an operation preceding the same OP_JZ, you will need to store the target address in the parser's stack. This can be done by assigning the next instruction to be generated (see macro INSTRUCTION_NEXT) to the variable **@$.begin.line**. To retrieve the value stored in some semantic action, use **@X.begin.line**, where X is the position of the semantic action as a symbol in the right-hand-side of grammar's rule. For instance, to retrieve an integer value stored in the stack by a semantic action occupying the second position as a symbol, use: **@2.begin.line**. This form of accessing the stack essentially replaces the simpler older form of **$2**.

3. (1.5pt) In grammar.y, complete the semantic actions for the while-do construct (see construct_while rule). Observe that the rule of this construct has 3 semantic actions. The first one precedes the condition evaluation. The second comes after the condition, but before the loop's body. The last semantic action takes places after recognizing and generating the code for the loop's body. The while-do construct potentially **jumps forward**, right after evaluating the loop's condition, to the first instruction following the loop's body whenever the condition is false. This means that you must use the new intermediate code operation OP_JZ (jump if zero). Since the jump target is to an operation that has not been generated at the moment that OP_JZ is being created, you will need to store the instruction entry associated to the jump, and complete it later in the third semantic action. The entry number to completed can be stored in the parser's stack. This can be done by assigning the last generated instruction number (see macro INSTRUCTION_LAST) to the variable **@$.begin.line**.

4. (1.5pt) In grammar.y, complete the semantic actions to implement the if-then and if-then-else constructs. See rule construct_if, which will use 3 semantic actions. The first one must generate a jump to the false-branch of the construct when the condition is false. Also store the instruction entry in the parser's stack as you will need it to complete the destination of the jump. The second action performs two tasks: i) generate an unconditional jump to potentially skip the execution of the else-branch, and ii) complete the jump's destination generated in the first semantic action. The third semantic action sets the target jump address for the unconditional jump created in the second semantic action. Be carefully to distinguish between **the last generated instruction entry** (see macro INSTRUCTION_LAST) and **the next instruction to be generated** (see macro INSTRUCTION_NEXT).

A number of test cases are provided (see inputs-outputs.tar.gz). If you write your own test cases, limit them to using only the **int** data type of our language. Also note that the current version of this compiler does not support <=, >=, == nor ! = (as in C syntax).

For convenience, a Makefile is also provided, but you are not required to use it. The Makefile will build two binaries, **simple.exe** and **simple-debug.exe**. The latter will output the symbol table, instruction table and a number of debug prints. Grading will be performed with **simple.exe** . If you need to add debug printing information, always enclose it between "#ifdef _SMP_DEBUG_" and "#endif".

To test a single input file, run: ./simple.exe < inputfile.smp

Grading will be performed based on the output of your compiler. Your output should match exactly the one in the .out files. Each .smp file has a corresponding output file. See the contents of the inputs-outputs directory (after decompressing the tar.gz file).

You can also test **all** the test cases of a single directory with the script **run-all.sh**. It expects the directory name to test.

Several online resources can be found in the web, for instance:

- `https://www.gnu.org/software/bison/manual/bison.html`

More resources can be found by searching for the key terms: yacc/bison parser generator.

Do not change any other file. Do not print anything to the output outside of the conditional compilation directives.

Each student should upload their modified grammar.y renamed to ABCDEFGHI.y, where ABCDEFGHI is the 9-digit code identifying the student (not the 4+4).