CS 3323 - Principles of Programming Languages
Fall 2021
Assignment 4

The objective of this programming assignment is to add basic control-flow functionality, and array manipulation to the `Simple` compiler being developed in this course (For future reference `Simple` is the name of our programming language). Instructions of how to proceed 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 Wednesday December 1st, 11:59pm. **LATE SUBMISSIONS WILL NOT BE ACCEPTED**.

Perform the following modifications to the compiler:

This version of the compiler introduces several new intermediate code operations: OP_LT, OP_GT, OP_LEQ, OP_GEQ, OP_JMP, OP_JNZ, OP_JZ, OP_CAST_FLOAT2INT, OP_CAST_INT2FLOAT. These new operations permit to implement control-flow constructs, one-dimensional arrays of type int and float, and also simple type-casts. See the file icode.cc, lines 326–457. Read their implementation to be sure you understand their semantics (i.e., the role of each argument). 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. They mainly affect 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).

Make the following extensions to grammar.y:

1. (2.0pt) 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 in the first semantic action. 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**. You must replace the three DEFINE_ME constants (Lines 153–167) by the correct expressions. You can inspire yourself from the `construct_while`.

2. (4.0pt) This version of the compiler adds support for one-dimensional arrays. See input and output test cases with the name prefix "arr0[1–3]" and "bubble-sort". Test cases with the suffix "-float" are the floating point version of the corresponding files without the suffix, while the ones without suffix use arrays of type int. Semantic actions for loading values from arrays can be found in Lines 443-474.

You are asked to complete all the TBD_ARG in lines 239, 241 and 264, which implement the code generation for storing values to entries of an array, and if necessary, type castings. As an aid, the correct opcode has been included in the calls to `itab_instruction_add`.

3. (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 file. No renaming is necessary.