Intra-function code generation

Description

Modify the parser to generate assembly-like code for each function in an input program. In another word, you are supposed to write a code generator, which, takes an original program as input, parses it, and then generates the code that can be represented by the modified grammar. Note, the modified grammar is not going to be used by your parser. It is for you (as a programmer) to understand what the output program should look like. The specific requirements are as follows.

- As in previous projects, your parser should ignore (but copy) "meta-statements" to the output program.
- The output program uses the same grammar as the input program with the following modifications. (The input program will be the same set of programs used in the previous two projects, following the original grammar.)
- Grammar modification for data declaration:
  - No data declaration is allowed except one array at program level and one array inside each function. You may assume only "int" type is used including the type of the function returning values.
- Grammar modification for assignment and expression:
  - <assignment> --> <id> **equal_sign** <operand> <operation> <operand> **semicolon** | <id>**equal_sign** <operand> **semicolon** | <id>**equal_sign**<func call>**semicolon**
  - <operand> --> **NUMBER | minus_sign NUMBER | <id>**
  - <operation> --> <addop> | <mulop> | <comparison op> | <condition op>
  - Change all mentioning of <expression> with <operand>, as in <id>, read/write statement, <func call>, <return>, and <condition>.
  - Now <id>s are array references.
- Grammar modification for control flow:
  - <if-statement> --> **if left_parenthesis** <operand> <comparison op or condition op> <operand> **right_parenthesis goto** LABEL **semicolon**
  - <goto-statement> --> **goto** LABEL **semicolon**
  - <label-statement> --> LABEL **colon semicolon**
  - LABEL is just another name for ID token
  - Do not use the previous definition of <if-statement>. Do not use <while-statement>, <break>, and <continue>. You will have to use the allowed constructs (e.g., if, goto, label) to implement what these disallowed constructs try to implement in the original program.
- Grammar modification for functions:
  - Function declaration and definition remain unchanged.
  - In <func call>, each parameter must be <operand> instead of <experssion>.
  - In <return>, use <operand> instead of <expression>.
- The output program should be compilable by the gcc compiler in our VCL image, and have the same behavior as the input program has. The name of the generated program should have "_gen.c" as the suffix (e.g., "foo.c" becomes "foo_gen.c").
- You may make the following assumptions about the input programs: (1) Array declarations can have only constant as the size. For instance, "int A[30];" can appear in the input program, but "int A[x];" or "int A[x+y]" or "int A[3+4]" will not. (2) Array elements can be passed as parameters of a function call, but arrays cannot. For instance, A is an array (which can be only one dimensional). A function call like "foo(A);" will not appear in the input program, but "foo(A[x+y]);" could. (3) Your can assume that the input program is valid if it passes through the scanner and parser. (4) We will use the set of sample programs posted in the previous projects for grading (except square.c). So as long as your submission works well on them, it is good. No need to support general cases beyond these test programs.

Three examples

Your generated code does not have to be exactly the same as those examples. There could be multiple legal forms for a given program. But your output should meet the requirement of the project.

original program 1
program 1 after intra-function code generation
original program 2
program 2 after intra-function code generation
original program 3
program 3 after intra-function code generation