Code Generation

Invocation

`compfs [file]`

where file is as before. **Wrong invocations will not be graded.**

Details

The program is to parse the input, generate a parse tree, perform static semantics, and then generate a target file. Any error should display detailed message, including line number if available (depending on scanner). Since P1-P3 were already tested, this focuses on testing P target code and and storage allocation. Testing will be performed by running the generated targets to assess if the runtime benhavior is correct.

The target name should be

- *file*.asm if *file* was the input. So if the input filename is file.m then the output filename would be file.m.asm

The program has 2 parts:

1. Code generation
    - 100 points
2. Storage allocation global option 30 (+20 for local)
    - **Global** - matches the global option in P3, all variables are allocated globally in the target language (persistent)
    - **Local** - this must be matched with the local option in P3; variables considered global in P3 are scoped in the entire program (persistent) and variables in a block are scoped in that block as in P3

Include README.txt file - must say Storage=Global or Storage=Local on the first line.

Otherwise global will be assumed.

Temporary variables can be processed as global scope even when doing local scoping for user variables.

Upon success, only display the name of the target file generated and no other output. Upon error and the error message, no other display should be generated and no target should be generated.

P4 Runtime Semantics

- **Basic semantics as in C - program executes sequentially from the beginning to the end, one statement at a time**
    - **means that the target needs to be generated from the tree left to right**
- **if statement is like in C**
- **while is like the while loop in C**
- **Assignment evaluates the expression on the right and assigns to the ID on the left**
- **+-* are standard arithmetical, / is integer division, unary : is negation**
- **All expressions are evaluated before being used**
- **Relational operators in RO**
    - **> < == != all mean the same as C**
    - **[=] returns true if the signs of the arguments are the same (with a 0 counting as positive)**
- **label statement places a label that can be jumped to directly using "warp"**
    - **Note that due to our static semantics, this identifier must be defined**
    - **labels should be unique (I should have had this checked during p3 but we did not, so you can assume they are).**
    - **You can assume that if there are any labels, that they are defined globally (again, ideally we would have checked this in p3)**
- **input reads in an integer from input and stores it in the identifier**
- **output outputs the given calculated expression**
- **IO reads/prints a 2-byte signed integer**
- **All data is 2-byte signed integer**

Data

- **All data is 2-byte signed integers**
- **Assume no overflow in operations**

Target Language

- **VM ACCumulator assembly language as given**

P4 Code generation Support

Code generation

- **The parse tree is equivalent to the program in left-to-right traversal (skipping syntactic tokens if not stored in the tree). The execution should be from the first instruction down. Therefore, the target needs to be generated performing left to right traversal of the tree so the target will also have top to down semantics**
    - **this traversal can follow static semantics traversal or be combined**

- if you did the global option in P3, you have to do the global option in P4. The P4 traversal will be after the P3 traversal.
- if you are using the local option in P3 and want to do global storage allocation in P4, the traversal in P4 can be after the P3 traversal, or they can be combined
- if you are using the local option in P3 and want the local option in P4, the P4 traversal is the same as in P3, the P3 code needs to be expanded to include now P4

- **When visiting a node - some nodes are**
    - **code generating - most likely only those that have a token(s)**
    - **not code generating - most likely without tokens but consider one at a time**
- **When visiting a node not generating any code**
    - **if no children, return**
    - **if children, continue traversal (likely left to right) then return**
- **When visiting a code generating node**
    - **some actions can be preorder, some inorder, some postorder, depending the node (determined by the label) and its semantics**
    - **if a value is to be produced, always leave the result in the ACCumulator**
    - **if a value was produced by another node you traversed to, find the result in the ACCumulator**
    - **if a value in the ACCumulator needs to be saved, store it in a new temporary variable with a proper name**
        - **to simplify, let us assume users never use variables starting with V or T so we can use them for temporary variables without conflicts with user variables**
    - **each node with the same label always generates the same code**
        - **regardless of location**
        - **may be one of multiple cases, one per production in BNF**
        - **the only difference may come from the token(s) found in the node (*e.g.*, different operator)**
- **At the end of the traversal, print STOP to target**

    - **to be followed by global variables+global temporaries in storage allocation**
        - **temporaries can be global or local on the stack if doing local storage**
- **Variables will require**
    - **variable creation upon definition - see storage allocation**
    - **variable access upon use - global option or temporary variable that is global just uses the variable name; local variable may need to be accessed on the stack; see storage allocation**

Below is code fragment that can be reused - how to create new names for temporary variables (start with T) and labels (start with L). The traversal does different things based on the node label. If needed, this generates new temporary variable or new label.

Keep in mind the generation of a variable or a label, and its actual use, may be separated by a call to another node that may generate new variables or labels - so these must be saved locally not globally.

```
static LabelCntr=0; /* counting unique labels generated */
static VarCntr=0; /* counting unique temporaries generated */
typedef enum {VAR, LABEL} nameType;
static char Name[20]; /* for creation of unique names */

static char *newName(nameType what)
{ if (what==VAR) // creating new temporary
    sprintf(Name,"T%d",VarCntr++); /* generate a new label as T0, T1, etc */
  else // creating new Label
    sprintf(Name,"L%d",LabelCntr++); /* new lables as L0, L1, etc */
  return(Name);
}

static void recGen(nodeType *nodeP,FILE *out) // recursive traversal
{ char label[20], label2[20], argR[20]; // local storage for temporary or label
  if (nodeP==NULL)
    return; // no nodes
  switch (nodeP->nodeId) // perform different actions based on the node label
  { case readNode:      fprintf(out,"\tREAD\t%s\n",nodeP->str);
                        break;
    case assignNode:    recGen(stats->child1,out);           /* evaluate rhs */
                        fprintf(out,"\tSTORE\t%s\n",nodeP->tokenP.str);
                        break;
    case ifNode:        recGen(nodeP->child3,out);              /* exprRight */
                        argR = newName(VAR);
                        fprintf(out,"STORE %s\n",argR);
                        recGen(nodeP->child1,out);              /* exprLeft */
                        fprintf(out,"%SUB %s\n",argR);          /* ACC <- exprLeft - exprRight */
                        label = newName(LABEL);
                        if (codeP->child2->token == ==Tk) {     /* False is ACC != 0 */
                          fprintf(out,"BRNEG %s\n",label);
                          fprintf(out,"BRPOS %s\n",label);
                        }
                        recGen(nodeP->child3,out);              /* dependent statements */
                        fprintf(out,"%s:\tNOOP\n",label);
                        break;
// etc.
```

P4 Storage Allocation Support

Must match that the target language offers.

- **All storage is 2-byte signed**
- **Storage needed for**
    - **input program variables**
        - **global variables need to be persistent, in target this is the variables declared wih storage directives after STOP**
        - **local variables will have storage allocated on the target stack**
    - **temporaries (*e.g.*, if accumulator needs to be saved for later use)**
        - **temporaries can be added to the global variables pool or allocated locally if using local scoping. I would suggest global. We can assume not to use variables named T# or V# in the source, reserving such names for temporary variables.**
        - **there is no need to optimize reducing the number of temporaries**
- **Global option**
    - **Storage allocation should follow the static semantics traversal (P3) and the and code generation traversal (in P4)**
    - **Issue the proper storage directive for every global variable of the input program (in STV of P3) and every temporary (created in P4 code generation), using the global storage directive in the target langage after the STOP instruction**
- **Local option**
    - **You must have P3 with the local option**
        - **Merge the P4 traversal with the P3 traversal (modify the P3 traversal)**
    - **Code generation discussed separately**
    - **Storage allocation**
        - **When in <vars> with an ID token (variable definition)**

            - **every push() in P3 must be now accompanied by PUSH in the target in P4**

        - every pop() in P3 must be now accompanied by POP in the target in P4
    - When in any node where n=find() was issued in P3 (variable was used) and if
        - n>=0 (the variable was found on the stack), then issue to the target
            - 'STACKR n'on reading access, the data moves to the ACC
            - 'STACKW n' on writing access, the data being written must be in the ACC
        - otherwise, variable is global so process as in the global option
            - temporaries can be generated as in the global option, or on the stack
            - <in> node with a token, resulting from "Read into a variable" in source statement, needs this
                - if find() <0 (global variable), you just write "READ TokenVariable" to the target
                - if find() >=0 (local variable), the inputted data must go to the stack location so you need to

                  Create a new Temporary T#

                  Write "READ T#" to the target

                  Write "LOAD T#" to the target to bring the read data to the ACC

                  Write "STACKW n" to the target to put the read data on the stack, where n=find(TokenVariable) in P3

- <R> node with identifier token needs if the variable is local - it needs to be fetched off the stack using "STACKR n", where n=find(variable) instead of "LOAD variable"
- <assgn> node needs similar treatment if the variable is local