CS445 Introduction to Compilers

Assignment 5
(Finally! The code generating assignment. Hooray!)
DUE: Sat Dec 14 at 5 PM PST

400 points

In this assignment you will convert your compiler into a full code generating compiler! This is the moment we have been waiting for. Congratulations! You have created a large complex program that will compile programs! You will be able to compile a C- program and... amazingly... it will execute on the TM machine! It is so cool to see the code actually run. This was our goal from the beginning of class!

 **Do not be fooled. This is a nontrivial homework. Do not put this off until you are doing end of semester projects and studying for tests because it is worth a boat load of points. If you get stuck ASK EARLY. Do not be in the position of figuring out how to do it with only 24 hours to go.**

To do this final stage, you will need to make modifications to the semantic section and to create a code generation section. The text below combined with what you learned in class will tell you what you will need to do to get this to work.

The results will be a C- compiler that will compile files for execution on the TM machine. The assignment will be graded by running your compiler on each test file. If there are no errors (warnings are OK) TM code will be generated. The code will then be given to the TM machine to execute. Your code running on the TM will need to produce the same behavior as my code on the TM. The TM code itself is free to be anything that produces the right output.

 **It is vital that you get the I/O routines to work and procedure calls to work or there will be no output and nothing to grade!!! I have tried to not do any crazy special cases in my code so it can be used as an example. Code is supplied in the test data.**

Command Line Improvements

1. **Add the -o filename option that lets the user specify where the tm code will go. This option can go anywhere in the command argument list. If it is not specified then the code will be written to the same filename as the source but with the extension changed to .tm. If no input file is specified the tm output will go to stdout. Some example invocations in a unix shell:**

```
c- fred.c- -o sally.tm     # compile fred.c- and put the results in sally.tm
c- -o fred.tm -p fred.c-
c- fred.c-                 # here the output will go to fred.tm
c- < fred.c-               # here the output will go to stdout
```

2. **Add the -P option that prints out the annotated syntax tree. This is just like the tree printed out for -p but also prints out the typing (local, global, param, static local), the size and location of each declared item. This can be easily done by passing a flag into your current printtree routine that prints this extra information when it prints the declaration nodes. This can be called after the semantic routine.**

The Semantic Analysis Changes

1. **You will need to augment the TreeNode to handle offsets of space on local, global, and static space for declared variables AND the size of the thing to be allocated. How to do this was discussed in class. (Feel free to ask if you want me to go over any or all of it again.) If you remember, you will maintain pointers for the place in global memory and the place in local (frame) memory in two offset variables in the semantic section. You will increment them as you allocate each variable and reset the *local* memory pointer within each function. The result will be that every declaration that requires space will store the offset of that space and the size in the TreeNode. A flag for whether the space comes from the global space or local space will also be set. You will also need to know if the variable is a parameter or not to know how to reference arrays that are passed into a procedure. You might have a flag for this. Mine records if the variable is a local, global, parameter, or localstatic variable. This will of course be used to tell you what offset register to use: 0 or 1 and how to load and store values.**

   The offset and size that is used for variables goes unused for all the rest of the nodes accept the function declaration nodes. We will use the size in the function node for storing the stack frame size. This can be set on exit of the code generation for the declarations in the function. The size can then be used to determine where the temp variables can be allocated. The offset in function nodes will be used later for storing the starting address of the function so that it can be looked up in the symbol table and we can jump to the various functions.

2. **The symbol table is already used to point to the TreeNodes so you can now find the offset of every reference to a variable when you arrive at them and look them up in the symbol table. You can now copy that information for use in those referencing nodes (e.g. variables in equations) as well during the tree traversal in the semantic analysis routine.**

3. **When you leave the semantic analysis section the symbol table will have been reduced back down to the globals section. If you freed the symbol table before... don't. It is valuable. You should pass symbol table of globals along with the end of the global space to the code generation section as described below.**

The Code Generator

If there are no errors (warnings OK), the Code Generation section will take the annotated tree, the symbol table containing the pointers to the global declaration nodes and the value of the top of the global space and generate the TM code as discussed in class. You might want to pass the size of the global space in a separate variable into the code generation code as well. You will need to know where the globals end and the function memory begins.

1. **I recommend you clone and cannibalize your semantic tree traversal for your code generator. Yes, it is another tree traversal, but you should be good at it now. :-) The code generator is written to traverse the tree generating the code by producing code snipits for the different nodes encountered. This may include code before, processing children, between processing**

children and after processing them.

I will supply you with the emit code you can use to write out TM code. If you want to modify this code or do it completely differently then great. Be my guest. It is similar to the code discussed in the book so you can use the book as a reference. This is your compiler.

2. **Figure out (or do as I suggested in class) how to install the I/O routines at the beginning and install the prolog code at the end. This task consists of putting in the prolog with the jump to main in the right place and placing the first frame right after the globals (as we discussed in class). The code for the I/O routines is copied at the beginning of your code so you know where each routine is so you can call it. Be sure the locations of the routines are recorded in the function nodes!**

3. **Although the offset, size, and type of all declared symbols is in the annotated tree is known as each are declared, the offsets for temporary space and the location in instruction space of the various routines can't be known until code generation time. You will need to manage a temporary offset for the temporary stack used as described in class for holding pending values in calculating expressions. You probably need a compile time pointer for the offset from the frame pointer for the temporaries.**

   You will need to put the address of the start of the code for each routine as you come across the declaration during code generation. Calls can then be constructed using the address found by symbol table lookups.

4. **You will need to initialize the size of the arrays. For globals this happens in the prolog code. For statics it is easiest, but not most efficient, to initialize when you initialize the local array sizes which is at the beginning of the compound statement in which they occur. This means that declarations will actually have actions.**

5. **You may do the simple backpatching illustrated in my code or you can save more information and do a more clever job of backpatching to produce more compact and or efficient code (the two objectives may not be the same). Remember that the optimizer is designed to clean up after us (provided we make it easy) in the next phase. The real objective here is to generate easy to optimize code.**

6. **At the end of the semantic phase you will know whether there is a routine named "main" or not. Sometime after that you should issue an error if main is not found. Here is the error:**

   ```
   ERROR(LINKER): Procedure main is not defined.
   ```

To save time these THINGS YOU DON'T HAVE TO DO FOR ASSIGNMENT 5: You don't have to gen code for:

- **strings**
- **array assignment or comparison**
- **foreach**

You still DO have to gen code for:

- **initialization**
- **static variables**

Code Examples and Source Code

Free Code

You should always grab code like this by doing a save link target or similar command in your browser rather than mousing up the code. Students report failures to correctly copy and paste from browsers.

- **emitcode.cpp**
- **emitcode.h**
- **emitcodeVarg.cpp allows for printf like formatting**
- **emitcodeVarg.h allows for printf like formatting**
- **Get the prolog code from any of the examples provided below.**

Example C- Source and Output

- **Here are some sample pieces of code for you to compile and run.**

Code Generator Example

Coming soon

- **Code Generator Code**

Submission

Homework will be submitted as an uncompressed tar file to the homework submission page. You can submit as many times as you like. The LAST file you submit BEFORE the deadline will be the one graded. For all submissions you will receive email showing how your file performed on the pre-grade tests. The grading program will use more extensive tests and those results will be mailed to when they are run.

 **If you have tests you really think are important or just cool please send them to me and I will consider adding them to the test suite.**
