C° Compiler Implementation Project
Phase 2.3: Code Generation for Procedures
Due: October 28, 2003

To complete a version of your C° compiler capable of producing runnable 34000 code, you must generate the instructions to handle procedure and function calls and generate the correct code before and after the bodies of procedures, functions and the main program. Your final output should be an assembly language program that accurately implements the C° program provided as input to your compiler.

Testing the Compiler's Output

To make it easy to process C° programs with your compiler, I will provide a short shell script named `cdimc` (along with lots of other new odds and ends in the `shared/434/Cdim/phase2.3` sub-directory). This script assumes your executable is named `Cdim` (as it will be unless you have changed the Makefile I provided). The `cdimc` script will expect the name of a C° source file as input. To make things look right, the source file's name should end with a `.c` suffix. The script will run the `.c` file through your compiler and then take what your compiler wrote to standard output and provide it as input to the 34000 assembler. To make it possible to use `#include` directives in the assembly code you output (I'll explain why you will need this ability later), the `cdimc` script will run your compiler's output through the C pre-processor, `cpp`, before sending it to the assembler.

The "final" output of this process will be a `tmem` file, which will be read as input by the `wc34000` interpreter program (or the `mice` interpreter if you decide to trust someone's microcode more than my interpreter). In addition, the script will leave the actual output of your compiler in a file whose name is obtained from the name of the input file by replacing the `.c` suffix with a `.s` suffix. Similarly, the output listing produced by the assembler will be stored in a file ending with a `.l` suffix (this file is actually more useful than the `.s` file because it shows in which word of memory each line of code is stored).

To enable you to keep your output code separate from error messages and diagnostics, I have written my code so that all output produced by printree, printdecldesc and DumpDecldescs is directed to "stderr". In addition, in case you want to keep the output that goes to standard output and standard error together, my routines start each line of output they produce with a ";". This will cause the assembler to treat such lines as comments.

How it All Starts

Execution will begin with the first line of code in the assembler file you produce. When this line is executed, the stack pointer will be set but no register will be pre-loaded with the address of the global variable area. To make it possible for you to load this value into a register (probably A5), the assembler puts the address of the first unused word in memory (the word after the last instruction in your code) in word 1 of memory. Thus, the instruction

```
MOVE    1,A5
```

is probably the first line of code your compiler should output on any input program.

Associating Labels with Procedure Entry Points

To enable you to generate correct code for calls, I have included a component in the declaration descriptor format used for procedures named `entrylbl`. It is intended to hold the "code label" placed on the first line of the actual code for the procedure. It will be used whenever you need to generate a JSR to the procedure.

The type of the `entrylbl` field is determined by the value of the #define `CODELBL`. If you do not include a `#define` for `CODELBL`, it defaults to the type `char`. However, if you have declared a special type to hold code labels, you can `#define CODELBL` to be the name of that type. For example, if your type's name was `codelabel`, you would simply include the define

```
#define CODELBL codelabel
```

somewhere before the `#include` for `symtab.h`. The definition of `codelabel` will also have to precede this include in most cases.

While I have included the `entrylbl` field in each procedure declaration descriptor, it is up to your code to set this field.

An Input/Output Library?

To make your compiler useful, you must provide a standard set of input/output routines. These routines should be named outnum, getnum, outch, and getch. They should provide a way to execute the corresponding 34000 instructions from a C° program. The routines outnum and outch will behave as procedures that expect one value parameter (of type integer). The routines getnum and getch will be integer functions taking no parameters.

To make it easy for you to add support for this input/output "library" to your compiler, I have done three things: 1) I have included code in the `init_symbtab` routine which creates declaration descriptors for these procedures and adds them to the symbol table; 2) I have provided you with code (in a file named `IOlib.c`) which includes procedures you can use to set the `entrylbl` components of the declaration descriptors for the I/O procedures; and 3) I have provided you with a file of 34000 assembly language code named `iolib.h` that contains the actual assembly language code for these procedures.

Like the code I gave you in `~tom/shared/434/Cdim/phase2/stmtgen.c` you may find that you have to modify the code in my `IOlib.c` and `iolib.h` files before you can use it. As a result, I will expect all of you to submit copies of the actual versions of these files you used. To make sure that this happens, you must include `IOlib.c` in the SRC line of your Makefile and `iolib.h` in the HDR line.

The routine provided in `IOlib.c` for setting the `entrylbl` components of each I/O procedure's declaration descriptor is called `initIOlib`. You should call this routine just before you begin code generation. It works by calling another routine named `initProcLabel` for each of the four I/O procedures. The `initProcLabel` routine assumes that the appropriate way to set `entrylbl` is to allocate a structure of type `CODELBL`, put the address of the allocated structure into `entrylbl` and then call a routine you must supply named `initLabel` to actually set the contents of the `codelabel` to a "new" label. You may have to write a special routine for initializing these labels, because you need the ability to specify the exact name associated with the label (the names used must match those in `iolib.h`).

The `iolib.h` file is not a C header file. It is a file of assembly language code to be included with the code you generate. Since `cdimc` runs the assembly code you produce through the C pre-processor, you can use this file by including the line

```
#include "iolib.h"
```

in the assembly language output your compiler produces. You will probably want to place this line either right after the code for the main program or after all the other code you generate.

Debugging Support

The one final complication is that I want you to include directives to the assembler that will enable it to pass enough information about your source program on to the interpreter to make symbolic debugging of the original C° program possible. The format of these directives, named `STAB` directives (for Symbol TABle), is described in the assembler handout. However, I have tried to make it possible for you to include these directives in your output without learning much about their format (and without writing much code).

To do this, I will give you yet another file of C code, `stabgen.c` and a new version of `main.c`. The new version of main.c differs from the old one in that it assumes you will give it a file name to read C° source code. This is important, because it enables your compiler to tell the assembler (and ultimately the interpreter) where the C° source code can be found. This is done by including an STAB-FILE directive at the start of your output file. The new `main.c` arranges to output this directive by calling one of the routines provided in `stabgen.c`.

I have given you the source code for `stabgen.c`, but I suspect you will not have to modify it. Nevertheless, you should make your own copy of this source file and add its name to the SRC line of your Makefile. It contains two routines that should provide you with an easy way to output all the `STAB` directives required.

The procedure `outputmainstabs` should be called just before you begin generating code for the main program. It expects a pointer to the root of the program syntax tree as its operand. It will print STAB directives describing all the types and global variables used in the compiled program. Obviously, it depends on your symbol table structures to determine this information.

The procedure `outputprocstabs` should be called once for each procedure in the source program just before you begin to generate code for that procedure. It expects a pointer to the `Nprocdefn` or `Nfuncdefn` node for the procedure as a parameter. It outputs STAB directives for all the parameters and locals of the procedure.

In addition to enabling the interpreter to provide debugging facilities for your C° programs, it is hoped that the somewhat readable nature of the STAB directives will make them useful when examining your compiler's output. In particular, for each variable, these directives provide a way to determine the displacement your compiler assigned to the variable. Thus, to some degree, they can replace the `DumpDclDescs` output.