CS143 Handout #32
Autumn 2000

# PP5: TAC code generation

**Due 11:59pm Thu Dec 7th**
(may be submitted at most **one** day late)

## The goal

In the final programming project, your job is to finish off your compiler by generating TAC code that can be run through Chowder and executed on the SPIM simulator. Finally, you get to the true product of your labor— actually executing SOOP programs!

Using our current syntax-directed translation model, you will add actions to the yacc productions to emit the necessary TAC code upon recognizing each language construct. Each action prints the TAC instructions to perform the operation, assign the variable, call the function, or whatever is needed. Thus, your job is to figure out where to put things in the parser and in what order. The output is a sequence of TAC instructions that can be fed to Chowder and converted to MIPS assembly. Because we are doing this in a single-pass syntax-directed translation, we have to be able to generate all the code in one continuous stream, without being able to look ahead or behind. Fortunately, the features of SOOP and TAC work well together and we can do this in a fairly straightforward manner.

Most students find this project to be around the same magnitude as pp4, perhaps a shade less time-consuming. It is also one of the most fun, it is an awesome feeling when you finally get to this stage where you can compile SOOP programs and execute them. Try not to get so distracted playing Blackjack that you forget to submit your assignment on-time!

## Starter files

The starting files for this project are in the **/usr/class/cs143/assignments/pp5** directory. You can access them directly on the leland filesystem or from the class Web site **http://www.stanford.edu/class/cs143/**.

The starting project contains the following files (the boldface entries are the ones you will definitely need to modify, you may need to make minor changes in other files depending on your approach):

| | |
|---|---|
| `Makefile` | builds project |
| `main.cc` | `main()` and some helper functions |
| `soop.h` | defines prototypes for scanner/parser functions |
| `soop.l` | SOOP scanner |
| **`soop.y`** | Yacc parser for SOOP grammar, accepts soop.ebnf |
| **`codegen.h/.cc`** | interface/implementation for codegen helpers |
| `declaration.h/.cc` | interface/implementation of Declaration class |
| `decllist.h` | interface/implementation of our provided decl list |
| `hashtable.h/.cc` | interface/implementation of our provided hashtable |
| `run` | simple script to run pp5, chowder, spim sequence |
| `scopestack.h/.cc` | interface/implementation of our provided ScopeStack class |
| `semantic.h/.cc` | interface/implementation for semantic routines |
| `tac.h/.cc` | interface/implementation of TacGenerator class |
| `type.h/.cc` | interface/implementation of Type class |
| `typelist.h` | interface/implementation of our provided type list |
| `utility.h/.cc` | interface/implementation of our provided utility functions |
| `samples/` | directory of test input files |

Copy the entire directory to your home directory. Run `make depend` to set up dependency information and then use `make` to build the project. The makefile provided will produce a compiler called `pp5`. It reads input from stdin and you can use standard UNIX file redirection to read from a file and/or save the output to a file:

```
% pp5 < samples/program.soop >program.tac
```

Once you have TAC code, you run Chowder to convert it to MIPS assembly that you can execute on the spim simulator. The executables for chowder and spim are in `/usr/class/cs143/bin`. You can either invoke them using the full path or add our bin directory to your path so you can invoke them just using the executable name. As a convenience, we provide the `run` script that will do all three steps in the sequence. You invoke it with one argument, the path to the SOOP input file:

```
% run samples/t1.soop
```

The output from the above command would be:

```
-- pp5 <samples/t1.soop >tmp.tac
-- chowder <tmp.tac >tmp.asm
-- spim -file tmp.asm

SPIM Version 5.8 of January 5, 1996
Copyright 1990-1994 by James R. Larus (larus@cs.wisc.edu).
All Rights Reserved.
See the file README for a full copyright notice.
Loaded: /usr/class/cs143/bin/trap.handler
hello world
```

We provide starter code that will compile but is very incomplete. It will not run properly and may even crash if given sample files as input, so don’t bother trying to run it against these files as given.

As always, the first thing to do is to carefully read all the files we give you and make sure you understand the lay of the land.

- The starter project is based on our solution to pp4. Much of the semantic analysis and error-detection has been removed in order to allow you concentrate on the task at hand. Thus, if you compile bad programs, it will not report most errors and will emit erroneous TAC code or perhaps even crash when compiling or executing. You do not need to test pp5 on bad SOOP programs (other than detecting a function/method used but not defined).
- We have removed doubles from the types of SOOP for this project. In the generated code, we treat bools just like our usual 4-byte integers, which evaluate to 0 or not 0. The only strings are string constants, which will be referred to with pointers. Arrays and objects (variables of class type) are also implemented as pointers. Thus all variables/parameters for pp5 are 4 bytes in size, which will help to simplify some things.
- The `Declaration` class has two new fields, the “offset” and “label”. For local/global variables and global function declarations, the offset field is not used (set to "NoOffset"). For instance variables, it is the offset in bytes to the field of the object, and for methods, it is the 0-based index into the virtual table (i.e., Vtable) for that object class. The `Type` class has been extended to assign offsets to fields as they are added to a class. The Declaration label field is only set for functions and methods and corresponds to its unique label assigned to that function. The label is set for you when the function declaration is created.
- The provided `TacGenerator` class has a variety of methods to output TAC instructions in the proper format. There is also `gTac` global already set up in soop.y for you to use.
- The `codegen` module is provided so you can separate the code generation helper functions into their own module.

## Code generator implementation

Here is a quick sketch of the tasks that need to be performed:

- Before you begin, go back over the TAC instructions and the examples given in handout #26, to ensure you have a good grasp on TAC. Also peruse the comments in tac.h to familiarize yourself with the TacGenerator class we provide to print out proper TAC instructions.
- You might want to start with variables and constants. Add actions to generate TAC var statements for all variables and parameters. Then add actions to load all constants.
- Generate instructions for arithmetic, relational, logical, and simple assignment operators. Note that TAC only has a limited number of operators. You have to figure out how to simulate the others given the ones we have.
- For function calls, you'll need to evaluate and pass the arguments and handle the return value. For the most part, method calls are like functions, so once you have functions working, you can get methods up and running similarly. There is a global DeclList of functions used that you need to keep updated as you process functions to assist with your "linker" phase below.
- Designators (array elements and field access) are a little more involved since they require computing offsets and dereferencing. By the way, SOOP uses the "array of arrays" implementation for multi-dimensioned arrays (see handout #27). Be sure to allow the "this." to be dropped inside a method, "this" is assumed for field access without an explicit receiver.
- Generating code for the control structures (if/while) will ensure you understand how to use labels and branches. Correct use of the break statement should work for exiting while loops.
- Finally, you will need to generate code for New, NewArray, Print and maybe a couple other things here and there. Just be sure you include everything!
- At the end of parsing, your "linker" (add some code that runs at the end of parsing) must verify that every function/method that was used was eventually defined,. The only error your pp5 must detect is a function/method that has a valid declaration, there are calls to that function that match that declaration, and no definition was seen. A declaration for an undefined function that is never used is not considered an error. There must also be a definition for the global function `main`. The error reported is `Function 'XXX' not defined.`

## Random hints and suggestions

Just a few details that didn't fit anywhere else:

- We are still using the C++ STL, which means that `make depend` will produce some warnings on the leland systems. Make (not depend) should run cleanly on the Solaris machines, but will print annoying warnings on the HPs. Either way, the built executable will run fine.
- You should not add any additional global variables beyond the few that we have provided.
- Keep soop.y uncluttered – define functions in the codegen module and just call them from your actions.
- You may find it convenient to re-arrange some of the productions to help with TAC generation. It's okay to change the grammar if necessary for your strategy.
- The separation between the compiler and Chowder creates a few limitations. One important one to be aware of is that Chowder will croak if a variable name is shadowed by another declaration in an inner scope. We will not test on such programs, and you shouldn't either.

## Testing your compiler

There are various test files that are provided for your testing pleasure in the samples directory. For each SOOP input test file, we also have provided the output from executing that program's object code under spim. There are many different correct ways of sequencing the TAC instructions, so it’s not that helpful to compare TAC outputs, but the runtime output should match. Be sure to test your program thoroughly, which will certainly involving making up your own additional tests.

Since we have removed much of the earlier error-checking, you should run your compiler only on syntactically and semantically valid input. The one exception to this is that your program should report an error for a program that uses an undefined function or method.

Congratulations, you have built a compiler in just 10 weeks! Did you ever think that would be possible back in September?

## Grading

This project is worth 90 points, and all 90 points are for correctness. We will be grading by running the object code produced your compiler.

## Optional extensions

Now that you have working compiler, the opportunities for fun extensions is wide open. Here are some suggestions for how to take care of all that unpleasant extra time you have on your hands during Dead Week. Figure that any bonus points can help improve your overall standing, especially if you had a little trouble here or there or lateness to make up for, but it will computed separated to avoid adversely affecting anyone who sticks to the basics. Here’s a few ideas to get you started, let your imagination run wild!

- Java is famous for its wonderful run-time checks that report and halt on various nasty errors that can’t be detected at compile-time. Wouldn’t it be great if SOOP had array bounds checking? Or reported when you attempted to send a message or access a field from a NULL object? There is special “Halt” TAC instruction you can use to generate to stop execution when you encounter a runtime error such as these. (After generating the code for it, you will understand why Java code runs slower than C…)
- How about bringing back the for and switch productions you added to the SOOP grammar as part of pp2 and adding code generation for them? How about pre and post increment or the ternary ?: operator?
- The logical operators && and || are fully evaluating in SOOP. Change the code generation to do short-circuit evaluation like C++ and Java (i.e. stop as soon as the truth of the expression can be determined).
- Add a pass-by-reference parameter mechanism to SOOP. Distinguish between value and reference parameters by adding a new keyword or punctuation symbol.
- Submit interesting programs written in SOOP. Like Peter Mork's Battleship, your legendary creation can live on to inspire further generations of CS143 students!

Whether or not you decide to add some bells and whistles, please make sure that what your submit will work with our scripts on the base requirements. You can include an alternate version with your submission or have a means of enabling special features with flags, just explain in your README file what we need to do to try it out. Be sure to include some sample files with your submission that show your extra features in action and tell us about those files in in your README.

## Deliverables

You may want to refer back to the pp1 handout for our general requirements about the submitted projects, electronic submissions, late days, partners, and the like. All the same rules apply here.

Today's thought on the election-that-wouldn't-die: "Half of the American people have never read a newspaper. Half never voted for President. One hopes it is the same half." — Gore Vidal