ECE251 Final Project, Part 3, Fall 2010

Patrick Lam

Due: Monday, December 6

Brief Overview

The final part of the project involves code generation for a small subset of WIG. This is a source-to-source translator from WIG to C. I've provided most of the code for you. In this document, I'll describe what you have to do to finish the code generator. My skeleton contains all of the methods that you need to implement. You just need to implement them. My implementation of these methods contains 141 lines.

Note that I've changed the command-line syntax of the `WIG` main class. To dump the symbol table, pass it `-symbol`. For pretty-printing mode (i.e. the part 2 behaviour), pass it `-verbatim`. The `-o` argument says that the next argument is the output file name. The `-baseurl` argument specifies a base URL, which appears in the script. The default behaviour of `WIG` is now to generate a `.c` file with the same basename as the input.

Submission and Marking

The best way to submit your project is by committing it to the SVN repository and also submit a `tar.gz` file to the external submission site. Once I've observed a handin to the external submission, I will try to pull your SVN and mark that; if I can't, then I will download and mark the submission to the submission site.

To mark your code, we will compare the output of your code with the output of our sample solutions on a small number of test cases (probably 3) and give marks according to how close your output is to a working solution.

Example

I'll start with an example. Here is the file `tiny.wig` from the test suite.

```
service {
  const html Welcome = <html> <body>
    Welcome!
  </body> </html>;

  const html Pledge = <html> <body>
    How much do you want to contribute?
    <input name="contribution" type="text" size="4">
  </body> </html>;

  const html Total = <html> <body>
    The total is now <[total]>.
  </body> </html>;

  int amount;

  session Contribute() {
     int i;
     i= 87;
     show Welcome;
     show Pledge receive[i = contribution];
     amount = amount + i;
```

```
    exit plug Total[total = amount];
  }
}
```

Compilers generate a lot of boilerplate code. The appendix contains the complete output from `tiny.wig`, including boilerplate.

Part-by-part description of code generation

The `tiny.wig` input contains three main parts: a set of HTML files, variable declarations, and sessions. We are not going to implement functions or tuples. I'll describe what you have to do for each of these parts. Note that the `CodeGenVisitor` has a field `w`, where it emits all of the code.

HTML

The `CodeGenVisitor` contains methods to visit the `HTML` class from the AST as well as three of the `HTMLBody`s. The `HTML` class emits C functions which will print out the HTML on standard output. In the case of an HTML with plugs, the C functions should take corresponding parameters. Your generator should handle the `Whatever`, `Input` and `Plug` HTML bodies by emitting `printf` statements with the bodies' contents.

Variables

One of the methods you need to implement is `printVariableDeclarations`. Query the symbol table (with the newly-introduced `getScope()` method) for all identifiers defined in a session. Print out C declarations for them, including their type (from the symbol table) and name.

Sessions

We will only handle WIG files with a single session and no functions. Our compiler implements the statements to print HTMLs: `exit` and `show`. These are mostly boilerplate, so I've provided most of the code for you. However, you need to write code in `leave(ShowStm s)` to load the variable names from the CGI parameters, when a `show` statement is receiving parameters. You also need to include visitor code for actually calling the `output` functions you defined earlier in the HTMLs part. This also includes code for passing plugs to `PlugDocument`s.

The one other general thing that you need to implement is code generation for saving and restoring all local variables, in the `saveVariableHelper` and `restoreVariableHelper` functions. These iterate through all of the local variables and emit `fprintf` or `fscanf` calls, respectively.

Expressions

You are responsible for emitting code for expressions. I've split out the code for that into a separate `ExpVisitor` class. I made the design decision that printing out sub-expressions requires a separate `ExpVisitor`, which we then read off the input of and combine with the parent `ExpVisitor`. While running a sub-visitor, push something onto the `silence` stack to avoid getting spurious output. While `silence` is non-empty, don't add anything to the output buffer. I'll describe how things work here in a bit more detail.

- Literals: Just emit them (if `silence` non-empty). I provide an example.
- Unary, binary, and assignment expressions: create a `ExpVisitor` and read off the output of that visitor. This gives you one or more strings. Combine these strings using the appropriate operator.

Extras

If you're really into this project, you can of course go beyond the stated requirements. I will not give you bonus marks, but I'll take note of the extra work you did. Let me know if you do this.

Appendix: Complete output from WIG compiler.

```
#include <stdio.h>
#include <stdbool.h>
#include <string.h>
#include <stdlib.h>
#include <time.h>
#include "runwig.h"

char *url;
char *sessionid;
int pc;
FILE *f;

void output_Welcome()
{
  printf("\n    Welcome!\n  ");
}

void output_Pledge()
{
  printf("\n    How much do you want to contribute?\n    ");
  printf("<input name=\"contribution\" type=\"text\" size=\"4\">\n");
}

void output_Total(char * total)
{
  printf("\n    The total is now ");
  printf("%s", total);
  printf(".\n  ");
}

int local_Contribute_i;
int main() {
  srand48(time(0L));
  parseFields();
  url = "http://localhost/~plam/cgi-bin/tiny";
  sessionid = getenv("QUERY_STRING");

  if (strcmp(sessionid, "Contribute")==0)
    goto start_Contribute;
  if (strncmp(sessionid, "Contribute$",11)==0)
    goto restart_Contribute;
  printf("Content-type: text/html\n\n");
  printf("<title>Illegal Request</title>\n");
  printf("<h1>Illegal Request: %s</h1>\n",sessionid);
  exit(1);

start_Contribute:
  sessionid = randomString("Contribute",20);
  /* i = 87; */
  local_Contribute_i = 87;
  /* show Welcome... */
  printf("Content-type: text/html\n\n");
  printf("<form method=\"POST\" action=\"%s?%s\">\n",url,sessionid);
  output_Welcome();
  printf("<p><input type=\"submit\" value=\"continue\">\n");
  printf("</form>\n");
  f = fopen(sessionid,"w");
  fprintf(f, "1\n");
  fprintf(f, "%i\n",local_Contribute_i);
  fclose(f);
  exit(0);
Contribute_1:
  /* ... receive []; */
  /* show Pledge... */
  printf("Content-type: text/html\n\n");
  printf("<form method=\"POST\" action=\"%s?%s\">\n",url,sessionid);
```

```
  output_Pledge();
  printf("<p><input type=\"submit\" value=\"continue\">\n");
  printf("</form>\n");
  f = fopen(sessionid,"w");
  fprintf(f, "2\n");
  fprintf(f, "%i\n",local_Contribute_i);
  fclose(f);
  exit(0);
Contribute_2:
  /* ... receive [i = contribution]; */
  local_Contribute_i = atoi(getField("contribution"));
  /* amount = amount + i; */
  putGlobalInt("global_tiny_amount", getGlobalInt("global_tiny_amount") + local_Contribute_i);
  /* exit plug Total[total=amount]; */
  printf("Content-type: text/html\n\n");
  output_Total(itoa(getGlobalInt("global_tiny_amount")));
  exit(0);
restart_Contribute:
  f = fopen(sessionid, "r");
  fscanf(f, "%i\n",&pc);
  fscanf(f, "%i\n",&local_Contribute_i);
  if (pc==1) goto Contribute_1;
  if (pc==2) goto Contribute_2;
}
```