csem: C SEMantic Compiler

Synopsis

"csem" is an assignment where we have to use a combination of Yacc (Bison) and the C Programming Language to make a "compiler" that can accept a subset of C, and convert it into intermediate code. The Yacc code and other various data structures were implemented for us in this lab, leaving only the actual functionality in C to write.

Approach

Getting Started

"csem" wasn't exactly trivial to begin writing. In fact, the hardest part about the assignment is getting started. Once past that, it was all good. I took the approach which involved me making several C test files, throwing it at the solution executable, and trying to mimic what it would print out. What was useful was that the functions in sem.c print out "csem: XXX not implemented", which made my life way easier in writing these functions.

Before even beginning the lab, I actually had to look at the source given and understand what exactly was given to us. We have 2 structs, "id_entry" and "sem_rec". We also have a multi-layered hash table (block-wise). And because sem_recs and id_entries can point to themselves, this also means that we have a singly-linked linked list data structure we can construct with pointers. On top of those data structures… I, at some point, made a lookup table for backpatching in sem.c to keep track of whether or not a branch variable was defined or not.

Order of Implementation

Note: This is going based on my GitHub history for the assignment.

The beginning

Ignoring my gradescripts for now, my first goal was to try to match the second example in the lab writeup (the first one was more complex anyways). To begin this amazing implementation cycle, I complained to Dr. Jantz about how bgnstmt was giving broken output. So he gave me a working

version of the function. After that, I went to add support for "id" and "op". Since it was early in the development, I actually needed to understand how sem_recs were being passed around. As such, I resorted to the "Dr. Plank" approach of spamming printfs (skillfully) which contained data on sem_recs that were passed into functions like "op1". I also did it for lookup results in the "id" function when looking for id_entry structs in that hash table. This is where I also started generating sem_recs for basically every action that I did in csem. So when I made a temporary variable, I made a sem_rec of the current temporary variable, and the data type, then returned it. The implementation of "con" and "set" were next. "rel" as well.

Making my "debugging toolbelt"

It was around that moment that I actually started running into issues like "what function is printing what?". Therefore, I made each function print out the function name alongside whatever it printed out. So in id()'s printf, I had:

```
//Print out information
printf(
          (p->i_scope == GLOBAL) ?
                  "[ID    ] t%d := %s %s¥n":
                  "[ID    ] t%d := %s %d¥n",
```

Of course, it ended up looking way messier than it should have, but I kept it because my printf calls were now powered by ternary statements and I thought that was cool.

At some point, I decided to implement my own macros for debugging and printing info. See it on the next page.

```
/*
 * ---------------------------------------------------------------------------
 * DEBUG LABELS
 *
 * Set certain macros to be defined for various debugging modes. This is mainly
 * to help with printing out data that may be useful when finding errors in the
 * intermediate code generation.
 *
 * Uncomment out any ones you want to work.
 *
 * LABELS:
 *     FUNC_LABEL - Print out the function name along with each operation.
 *                  Useful because sometimes I don't know where the hell a line
 *                  of code breaks. This just says what function the program is
 *                  at when printing (well, the first 7 characters of it).
 *
 *     FUNC_NOTIM - Forces the functions to not be implemented and return the
 *                  "sem: X not implemented" message to be printed to stderr.
 *
 *     FUNC_SEPAR - Forces the functions to spit out their function names
 *                  "before" actually printing out what the function does. This
 *                  is to print out a "stack trace" of what functions are being
 *                  called, and what order they are being called in.
 * ---------------------------------------------------------------------------
 */

//#define FUNC_LABEL
//#define FUNC_NOTIM
//#define FUNC_SEPAR
```

Here's what each does:

- FUNC_LABEL – Prints out the [FUNC] tag before every printf statement (On the same line).
- FUNC_NOTIM – Forces every function to not be implemented.
- FUNC_SEPAR – In my code, there is a "FUNC_LOG" function. If this flag is on, then that function will print out the function name *before* the printf statement. It will also print the function even if it doesn't print anything, so you can get a "stack trace".

Backpatching and Function Calls

So now that I have my toolkit to debug the lab comfortably, it was time to get back to writing my functions. It was at this moment that I started to implement "backpatch". I'm not going to lie. At first, I thought it was going to be a complicated function, but it ended up being a printf statement.

To demonstrate my new-found backpatching abilities, I wrote "m", "n", "doif", and "doifelse" (which was sort of like how the slides did them, but..... Yeah it was different). I also wrote "string" in preparation of function calling. That led to "exprs" and "call". "exprs" is the first instance I ran into where I had to construct a linked list. So instead of me making my own, I abused the pointers in the sem_rec struct to make one there (yes, I had to verify that this actually works). The twist to it was that, in call(), the linked list was backwards. So I had to malloc a struct sem_rec**, to keep track of

the elements, then traverse the list backwards to print out the correct argument numbers and types. This is also when I thought "fi" was hardcoded as "Function Finish" (only to be corrected because of "ff" later on).

My call function supported nesting instantly. But then I noticed that I had to consider casting temporary variables. This is because, what if I decided to throw a double into a function that only accepts ints? And what if I tried to use a function that returned doubles, to assign a value to an integer variable? So I had to go back to pretty much every function that actually needed casting (and note how "con" returns a T_INT without being "|="'d with T_ADDR) and implement it. It took a lot of testing actually to get right, but eventually I got it matching the solution code.

Loops

So moving on, we have backpatch, as well as a few "do" functions complete… so how about some loops? I first did dowhile and dofor, which were both *straight off of the powerpoint slides* for "Intermediate Code Generation"… almost. I had to change a few things to make sure it matched the solution's output. It worked as intended… except whenever I had more bugs with casting come to haunt me. I fixed them and everything was okay. "dodo" wasn't hard either.

Logical operations

So what about "and", "or", "xor", etc in if statements? Yes, they exist, and yes I had to make mine support it. This was late into development to where I was just trying until I got it right… except I didn't get it right. Then I found this (Page 411 in class textbook):

The translation scheme is in Fig. 6.43.

1)	$B \rightarrow B_1$ **||** $M\ B_2$	{ $backpatch(B_1.falselist, M.instr)$; $B.truelist = merge(B_1.truelist, B_2.truelist)$; $B.falselist = B_2.falselist$; }
2)	$B \rightarrow B_1$ && $M\ B_2$	{ $backpatch(B_1.truelist, M.instr)$; $B.truelist = B_2.truelist$; $B.falselist = merge(B_1.falselist, B_2.falselist)$; }
3)	$B \rightarrow$! B_1	{ $B.truelist = B_1.falselist$; $B.falselist = B_1.truelist$; }
4)	$B \rightarrow$ (B_1)	{ $B.truelist = B_1.truelist$; $B.falselist = B_1.falselist$; }
5)	$B \rightarrow E_1$ **rel** E_2	{ $B.truelist = makelist(nextinstr)$; $B.falselist = makelist(nextinstr + 1)$; $gen('\texttt{if}'\ E_1.addr\ \textbf{rel}.op\ E_2.addr\ '\texttt{goto}\ \_')$; $gen('\texttt{goto}\ \_')$; }
6)	$B \rightarrow$ **true**	{ $B.truelist = makelist(nextinstr)$; $gen('\texttt{goto}\ \_')$; }
7)	$B \rightarrow$ **false**	{ $B.falselist = makelist(nextinstr)$; $gen('\texttt{goto}\ \_')$; }
8)	$M \rightarrow \epsilon$	{ $M.instr = nextinstr$; }

So what happened? *I flat-out copied it and it worked*. This includes ccand, ccor and ccnot. ccexpr was funny because I just compared the constant "0" to "e" with the "!=" operator and it worked.

labeldcl and dogoto

This… is where things didn't go so well for a while. To implement gotos, I had to perform a small "hack" on the id_entry class, where I used its "i_width" variable to tell if the goto has actually been defined yet. I also had to use dclr and lookup in order to properly implement labeldcl. However, it only supported gotos to places *that already existed*… So what exactly is the deal here?

…Backpatching

So we do not know where the goto will go yet, because the label wasn't defined. Therefore, we place it in a temporary goto spot and then, when the goto is defined and known, "patch it up". It was somewhere around this point that I realised that "backpatch" isn't just a printf statement. It's actually a loop. So if p->back.s_link != NULL, then traverse it and print it out as well until the pointer is NULL (as if it was a linked list). Furthermore, I realised the importance of implementing ftail with leaveblock, and also startloopscope and endloop scope. Because the blocks are important in that hashtable to tell what labels exist and what don't in certain block levels.

…And I forgot to implement indx, so I did that too.

The gotos still had a few bugs, as I proceeded to make gradescript test cases for them. I ended up failing my own gradescript. So back to reimplementing dogoto and labeldcl… I managed to fix my bugs after reworking my block management almost entirely (added "endloopscope" and "leaveblock" wherever needed).

dobreak and docontinue

So I thought I was done… and then I scrolled across the wonderful functions "dobreak" and "docontinue". So, just like the previous functions "labeldcl" and "dogoto"… these functions ended up breaking my program and forced me to rewrite how backpatching worked. Also, these functions were the only functions which forced me to implement a lookup table for branched labels that were backpatched.

The policy was simple. Whenever we "goto", and we don't know where to go, so we set the value in the lookup table to 0. When we know the value, change it. The only reason why I had to do this

was because the structs that were chained occasionally repeated and showed gotos that were already backpatched. So I simply checked if those were "0". If they were not, print. Otherwise, ignore. After that system was set up, I then was able to implement "dobreak" and "docontinue". However, this also meant that I had to add in additional statements between my backpatch calls in my "dodo", "dofor" and "dowhile" functions to match the solution executable. At that point, I just tried stuff until I eventually got it.

The final function…

Alas, I implemented "dobreak" and "docontinue"… so I thought I was done for real this time. However, I forgot to implement "opb". Initially I thought to just copy and paste what I had for the other op functions, but then I realised that the precedence is flipped for this particular function. So what did I do? I just reversed how the function worked and it matched the solution executable. After all of this, I moved on to making my gradescript test cases.

Debugging

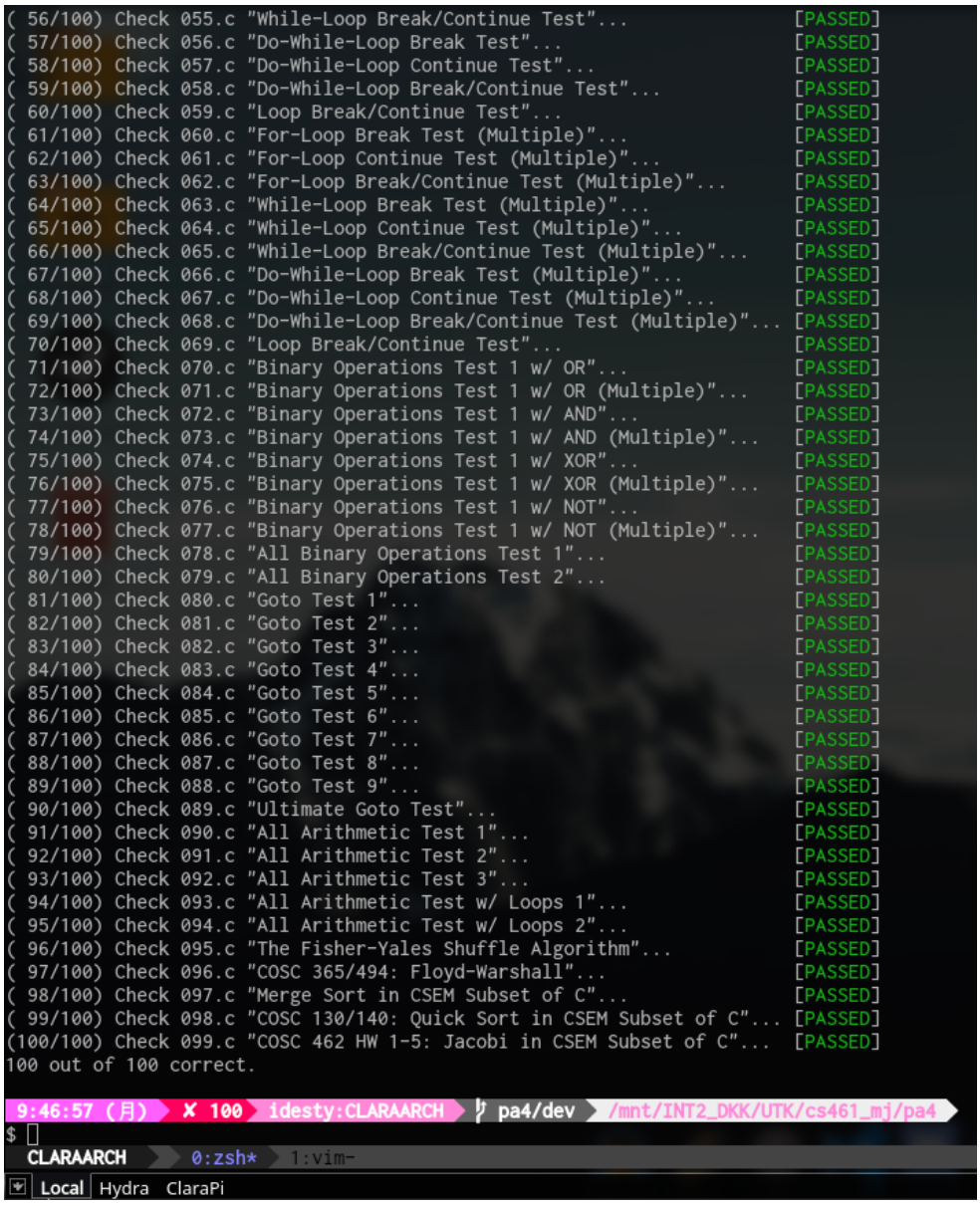
Gradescripts

Unlike the last assignment, I decided to type out the gradescript test cases for this assignment by hand. I got several complaints from students who used my gradescripts for PA3 as they were too difficult. So I made it much easier to use this time around, and also made it way easier to debug. Each test case shows what functions are needed to pass it. And also a lot of test cases build upon the previous, usually in a series of 1-to-3, or 1-to-5. A gradescript test case looks like this:

```
1  /*
2   * Variable Assignment Test 1
3   *
4   * Required Functions:
5   *     bgnstmt, con, fhead, fname, ftail, id, set
6   */
7
8  double i[8];
9  int a;
10
11 main() {
12         int y;
13
14         y = 2;
15 }
```

"005.c" in "gradescript/input/".

A few other students who have attempted the assignment found these gradescripts much more useful than PA3's, and were able to get 100/100 shortly after finding bugs in their code. Here is an example screenshot of what it looks like to run the gradescript:



It prints out the test case file name, and the type of test it is running (By using "sed" on the second line of the file). This way, students can know what exactly is failing *even before looking at the test case file.* Also, the last few test cases were specifically constructed to (probably) auto-pass if the students have passed the previous test cases. The last 5 were based on algorithms other classes rewritten by me exclusively for csem. I hope you enjoy, considering how long it took to write them all by hand. 😉

Conclusion

Difficulty

Dr. Jantz said it'll be more of a challenge… and he wasn't lying. It was a challenge……. To start. csem isn't hard whenever you understand what exactly is going on in the code that was given. The only hard part is simply starting on the assignment. Now, I didn't enjoy having to fix issues with backpatching almost three times, but that was more tedious than it was difficult. Needless to say, I'm glad it works.

Usefulness

The one thing I wanted out of this class was to be able to know how to write my own scripting language ("CN_SCRIPT"). I think, with the knowledge I have learned so far in this class, I can make an interpreter for this new scripting language and actually see it work wonders. That'd be really nice actually. I enjoyed the assignments. Thanks for the "*challenge*". 😉

~Clara