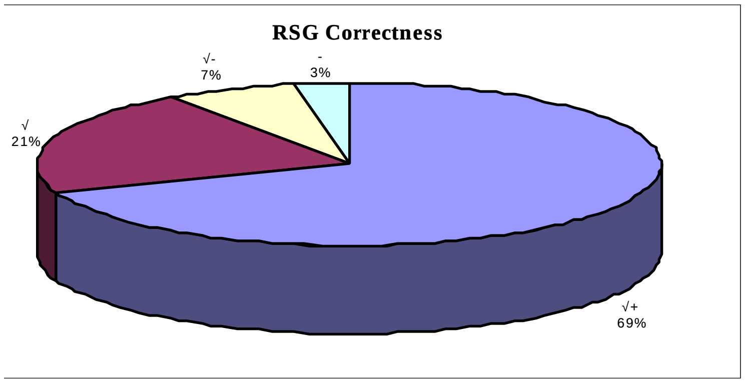
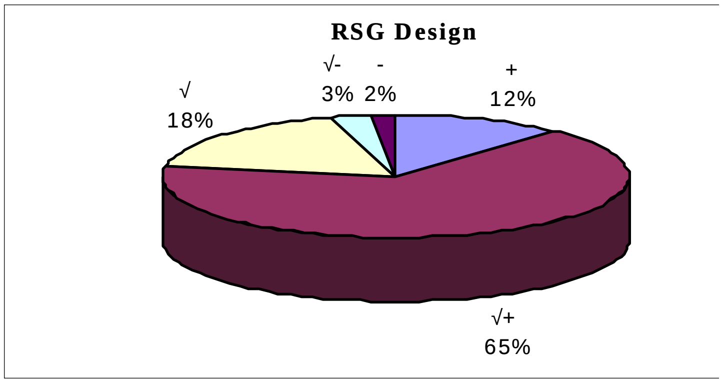
CS107 Handout 48
Autumn 2000 November 20, 2000

HW1c RSG Feedback

Feedback written by Julie and revised by Jerry.

We'd temporarily diverted our staff attention to grading the 107 midterms and allowing the TAs to study for their own exams, but now we're back to assignment grading in full force. My understanding is that RSG feedback was emailed out over the past week. TAs have all reported their distributions to me, so I know that all of them are done with the grading. Look at all these check plusses(!):





General Comments

Overall, these programs were very, very good. We gave many perfect and near-perfect scores in functionality and lots of check plus design scores. Those of you who killed yourself over hw1b details, thoroughly testing and debugging your ADTs and working through the memory issues in the client, found that it really paid off when writing the RSG. Congratulations on a job well-done!

Those who never quite got hw1b working had it come back to haunt them for a second round. Although most of you prevailed in the end, I realize that must have been a frustrating experience. Unfortunately, this is the way that programming in the real world often works, so you will want to develop a strategy for dealing with this sort of staged development. In the past, the advanced C program has gone out in just one assignment, with no intermediate deliverables. You can imagine the difficulties that can result from trying to pull together all three parts at once. Our idea in breaking it up was to give you a chance to focus on just one part of the problem at a time. And by providing test code and compiled binaries, we hoped that you would be able to identify your problem area and concentrate on just that. This strategy somewhat breaks down when you have multiple problems lurking in there.

Here's the overall distribution of scores for design and functionality. I think you'll agree that the programs must have been very nicely done for these pie charts to be so check-plus heavy.

The single most important issue of the entire RSG

By far the most common error in the programs was the same one we saw in the hw1b clients: incorrect interpretation of the `void*`. You pass to `ArrayAppend` a pointer to that element, `ArrayNth` will return a pointer to the element, all the callback functions (`Hash`, `Compare`, `Free`) take a pointer to an element, etc. This is true even if the element type is already a pointer (such as `DArray` of `char *` or `DArray` or `DArrays`). Understanding this is key to use your ADTs correctly.

Data Structure & Design

- There was pretty much just one correct data structure:
    - A Grammar is a `HashTable` of Definitions
    - A `Definition` is a `struct` with a `char*` (the non-terminal name) & a `DArray` of `Productions`
    - A `Production` is a `DArray` of `char*` (a sequence of terminals & non-terminals)

  There were some odd designs that used the `Hashtable` and `DArray` in rather non-sensical ways. Data structures always come out best when you think about the problem you're trying to solve and carefully choose the right design for your needs. The `Hashtable` is designed for fast lookup, and thus a good choice for managing the association of non-terminal to its possible productions. The productions themselves are an unordered collection, handled nicely by the `DArray` since you don't have a fixed size known in advance. And each production consists of a sequence of pre-parsed tokens that you need to keep in order, and later iterate over to print or further expand, another good use of the `DArray`.
- Some students "rolled their own" linked list or array handling for the lower levels, rather than use a `DArray`. We deducted points for reworking the problem that `DArray` already solves. Similar, some students didn't seem to want to use their nicely designed `Scanner` object to parse the grammar files and made other arrangements. Solving a problem a second time is inadvisable not just because it adds effort and little or no value, it opens up many more opportunities to introduce error into the program.
- Although we recommend against this in the handout, some students tried to cram all of the words in the line of a production into a single `char *` and then tried to parse it out of the string during the expansion phase. This was incorrect not only from a design point of view, but also because it necessitating re-parsing these "superwords" repeatedly during the expansion phase. The most compelling reason against this, though, was that means you couldn't use the `Scanner`, since it works on files, not strings. Thus it meant writing a bunch more code, and probably hitting a nasty wall when you found that `strtok` is not re-entrant. Be sure to try using the tools you already have before you jump into writing anything new!
- It was a better decision to include the angle brackets as part of the non-terminal name than remove them. It made the file easier to parse, there was no special handling for non-terminals versus terminals, and distinguishing between the

two merely involved checking the first character. Removing the brackets required more complicated handling and necessitated some other boolean to be managed to distinguish non-terminals from terminals.

- Some students kept a count of how many elements were in the `DArray`, i.e. redundantly storing that number instead of just using `ArrayLength` to get count as needed. Not only is this wasteful, it can be error-prone if you aren't vigilant about keeping the two lengths in sync.
- When storing the words in your data structure you should have been making heap copies of just the needed size and storing a pointer to them in the `DArray` rather than storing each word in a fixed-size character buffer. Since the max token size was quite large, using that bound as your storage size meant wasting a lot of space for all the shorter words.
- (A repeated comment from hw1b): It is best to avoid extra levels of indirection when you can. For example, in the grammar hashtable, it is much preferable to store `Definition structs` themselves as elements, rather than pointers to `Definition structs`. Using pointers introduces an extra dereference on every access, and requires that you dynamically allocate many small structures, fragmenting the heap, and means you have to be careful to free them all when done. For the ADTS, though, you had to work with pointers, since that is all the client has access to.
- (Another repeat from 1b): In a similar vein, if you only need a variable temporarily (i.e. for the lifetime of a particular function) and its size is known at compile-time, you should allocate space for it on the stack, rather than dynamically allocating it at the beginning of the function and freeing it at the end. Using dynamic allocation is less efficient and more prone to correct problems (`malloc`ing the wrong size, failing to free, etc.) Also, you should realize that both the `DArray` and the `HashTable` copy the element you pass it, so you do not need to keep a copy around (i.e. it is okay for element to be on the stack and go away).

Parsing

Most students were able to do this part, perhaps with a few quirks in how they parsed the file. However the parsing was often done with a lot of rather ugly and convoluted code. The file could be read rather cleanly if you spent some time carefully designing your approach. We found a strong correlation between the parsing code's decomp/readability and its correctness.

- The parsing phase had three natural loops in it, reading all definitions, reading all the productions in one definition, and reading all the words in one production. Each of these is a prime candidate for a separately decomposed function. Parsing the entire file in one big function was much too cluttered to read, and I'm sure it wasn't much fun to write and debug. Decomposition is one of your best tools for handling a complicated task, don't abandon it.
- To pull apart the grammar files, you basically just needed to set the `Scanner` loose on the job, using whitespace as delimiters and allowing them to be discarded. Once you're humming along, it's merely a matter of noting when

you get to the special tokens that mark the start and end of the various components and dealing with them appropriately. There really was no need for anything more complicated than that.

- Most students used "`SkipXXX`" to pass over the junk outside the definition braces and up to the beginning of a definition or non-terminal. This is the desired solution, since parsing them into tokens and discarding them is less efficient.

Recursive Output Generation

If your program was able to parse and construct the grammar data structure, it most likely was able to successfully generate random output.

- The output generation phase was supposed to rely on a recursive function. Just about everybody got that part. The choice of where in the chain of events to recur was one on which many made different decisions. There were several designs that are all essentially correct.
- The cleanest solutions used `ArrayMap` to process each word in a production.
- We expected you to use `TableLookup` to find the non-terminal's definition, and to detect and flag the error when a non-terminal was used that had no corresponding definition.
- The start non-terminal should have required only minimal special-case handling, since expanding it took the same steps as all later expansions.
- The `<ctype.h>` macros test a single character to determine its character class. In addition to the familiar `isalpha`, `isdigit`, etc., there is also an `ispunct` macro that came in handy for testing whether a word began with a punctuation mark and concluding whether to output a leading space. We didn't expect that you try to enumerate all the variations of special cases for when a space was appropriate or not, something simple that is mostly right was just fine. When in doubt, ask us or observe the behavior of the sample program.
- Some programs wrapped output in weird ways because they intermingled use of our `PrintWithWrap` functions with calls to `printf`. This is no big deal to us, but it was very easy to fix and I am surprised that your sense of pride and curiosity didn't make you want to get to the bottom of the issue and tidy up your output.

Freeing Data Structures

Most people who attempted it got most of it, but perhaps missed a detail or two. Purify is exceptionally handy for pointing out the leaks and helping you plug them.

- The basic gist was to free everything that was dynamically allocated. An ADT must be freed by the corresponding ADT function (Table: `TableFree`, DArray: `ArrayFree`, etc.) and any internal pointers within the stored data must first be freed by the client's element free callback function (assigned at `TableNew`, `ArrayNew`).

Readability & Style

- Don't use a `while(TRUE)` loop if the termination test with a break is the first statement of the loop body, just place the test directly in the `while` statement!

- Watch for unnecessary and redundant casts. For example:

```
int ch;
ch = (int)SkipUntil(s, "{");
```

  The `SkipUntil` function is declared to return `int`, so the cast is completely unnecessary. Moreover the cast is actually dangerous, because its presence completely overrides the type-checker. If `SkipUntil` was later changed to return a different type, the cast would prevent the type-checker from bringing the resulting mismatch to your attention.

- Some students are still having trouble developing a naming convention that is meaningful. To them, I recommend re-reading the Documentation & Style handouts. Avoid variables named directly after their data types, this isn't really that helpful:

```
int *intPtr;      // poor
HashTable table;  // poor
DArray array;     // poor

DArray productions;  // good
HashTable grammar;   // good
```

- Structs defined with only one member are of questionable value. For example:

```
typedef struct {
   DArray words;
} Production;
```

  Although I can admire the sentiment of wanting to provide distinct type names for the different `DArray`s you had floating about, a `struct` with one member is an oddity and should be avoided. If you can't live without different type names for various uses of the `DArray`, you might introduce synonyms like this:

```
typedef DArray Production;
```

  This doesn't bring in any type-checking (any `DArray` is type-compatible with any `Production`), but it might help you keep them straight. However, this comes at some other cognitive cost to the reader, who now has to manage more type names and keep track of what they really are. I think descriptive variable names are the best way to indicate which `DArray`s are which.

  You might recall that the `LexiconADT` from the exam wrapped a `struct lexiconCDT` around an array of `DArray`s, so you might ask why it was okay there and not here. The difference is that the `ADT` abstraction versus implementation paradigm forces us to bundle implementation state into a `struct CDT`, regardless of how simple and compact it is. Otherwise, we'd have to `typedef` the `lexiconADT` to be an array of 676 `DArray`s in the interface file, and that would have exposed too much.

Miscellaneous

- You should test on at least all the given input files, even if you don't create any of your own.
- Make sure that the electronic submission of your project contains a `Makefile` and all the files necessary to compile your project without warnings.