Appendix A

REFLECTIONS BY ONE GOOD STUDENT

These example milestone reports are courtesy of Clint W. Smullen IV, who took a course based on this text in the fall of 2005. These notes are provided to give students an idea of what a report should look like.

The reports are Clint's; I have removed code and drawings that would trivialize the projects.

A.1 MILESTONE I

A.1.1 Metacognitive Comments

I first began to learn Gforth by translating extremely simple expressions that I know in C into Gforth, such as: 1 + 2 into 1 2 +. I then proceeded to look at how to manipulate the stack using ., .s, DROP, ROT, etc. This was followed by learning how to use floating-point arithmetic and the large differences between the manipulation of integer and floating-point numbers. It was not until after class on Monday that I knew how to use variables. At this point, I was able to translate the first two cases of expressions into Gforth, except for the integer exponentiation. To learn how to create words and utilize conditional expressions, I wrote a new word in Gforth to compute integer exponentiation. This heavily used the stack manipulation functionality which I had previously studied. From studying these various aspects of Gforth and translating instructions from a language I know well to this new language, I have learned something of the pathway and issues that will arise in the process of writing a compiler.

A.2 MILESTONES II AND III

[Authors note: In the semester that Clint participated, there was only one report covering both the scanner and parser.]

A.2.1 Designing the Scanner

Starting with the basic regular expressions for each class of token, I developed finite state automata (FSAs) that process these inputs. The basic classes of input are integers, floating point, strings, and names. The name token type also incorporates booleans and files. These FSAs were extended into finite state machines.

A.2.2 Implementing the Scanner

I programmed my scanner to read chunks from the input. And scan through the chunks looking for matches. First, it looks to see if the current input character is either a left or a right square brace, then it checks for an opening double-quote character. If it finds a double-quote character, then it will begin parsing for a string. Otherwise, the scanner extracts the longest possible input token from the input and tries to match it to an integer, float, or name. The C Standard Library functions `strtol` and `strtof` are used to check the input for being an integer or float. This ensures that the scanner "does as C does," when it comes to integers and floats. If the input token is not entirely an integer or a float, then it must be a name. At this point, I directly used the name token FSM to check this assumption. Once a set of data from the input stream is matched to the input, the data is stored into a token structure along with the matched type and returned back to the caller.

A.2.3 Designing the Parser

I designed my parser around the simplified grammar that was discussed in class. This grammar moves the vast majority of the complexity inherent in the original grammar from the syntactic analysis that is occurring in milestone III, to the semantic analysis that will occur in later milestones. This simplified grammar serves only to rebuild the tree from the linearized version which is given as input to this compiler. The two necessary components center around the productions $\{S, T\} \rightarrow [\ T\ ]$ and $T \rightarrow T\ T$. The former of these two productions represents parsing the child of a node while the latter represents parsing a sibling. In a more compact form, the grammar can be made to show these two possible paths in parsing.

A.2.4 Implementing the Parser

In programming my parser, I used the compact form of the grammar almost directly. The function that does the bulk of the parsing, called `childParser`, first checks for a left bracket token. If it finds one, then it recursively calls itself again until it sees a right bracket token. If it does not see a left bracket, then it directly saves the token that was read into the tree.

This function by itself implements only the productions starting at *T*, so, to complete the parser, I added another function, called `parser`, which calls `childParser` once and then checks that the node returned is a list and not an atom. These two functions form the same functionality described in class as the double-recursive algorithm, with the modification that one of the recursions is replaced by an iteration. Two functions are still necessary to obtain the correct input language.

A.3 MILESTONE IV

A.3.1 Designing the Symbol Table and Type Checker

First, it was necessary to decide upon what data will be stored in a symbol. For function symbols, the name of the function in both SOL and Gforth are necessary, as is the return type and the types of all of the arguments. Variable symbols need the name, the type, and a value for static variables such as true, false, stdin, etc. The symbol table structure is an array of pointers to symbols. There will be two arrays for each type of symbol: variables and functions. Though this will require more duplicated code, it simplifies the design, because there is no ambiguity about what the type of a symbol is.

Starting with the premise of a hash table, I looked at different possibilities and decided on using open-addressing with quadratic probing. By keeping the hash table less than half full and using prime table sizes, it is always possible to find a hashing location with quadratic probing. Following onto this, it was necessary to design hash functions to hash the symbol structures as well as a node from the parse tree and have the hash values match. [Author's note: this is completely counter to the requirements!]

I decided to use the string hashing algorithm described in Mark Allen Weiss's *Data Structures and Algorithm Analysis in JAVA*. This algorithm is similar to the algorithm used in the Java libraries for hashing. Although this algorithm alone is sufficient for computing the hash values for variable names, it is not quite enough to hash the complete input specification for a function. To incorporate the effect of arguments, I decided to extend the use of the hashing algorithm above by additionally hashing the types, in numeric form, at the end of the name string. When running a parse tree through the type checker, all of a function's arguments will have been typed before looking it up in the symbol table, so it is universally acceptable to use this method. When hashing a symbol, all of the types are immediately available and, when hashing a parse tree node, the types of each argument will have already been determined.

The type checker is a bottom-up algorithm, as discussed in class, which checks all of the children of a node before performing type checking on

the node itself. Upon encountering a token node in the parse tree, the type checker will use its type directly for non-name tokens or attempt to look its type up using the name given. To type check a list node with the contents $a_0\ a_1 \ldots a_n$. The types of the children $a_1 \ldots a_n$ will first be checked recursively. An error is thrown if a_0 is not a name token, otherwise, the symbol table is searched for a symbol matching the name a_0 with matching the count and type of $a_1 \ldots a_n$. If no symbol is found, then an error is thrown, otherwise a pointer to the symbol is saved into the parse tree. This way, both the name of the Gforth library word and the return type are immediately available without any need to add or alter the data in the tree.

A.3.2 Implementing the Symbol Table and Type Checker

I implemented the structure of the symbol table exactly as designed along with hashing routines, hash table resize and rehashing routines, as well as routines to print out the contents of the symbol table. The hashing routine was implemented as designed, with the modification that when combining on the effect of function types, the previous iteration's value is multiplied by 67 instead of 31. This gave improved distribution and reduced the overall number of collisions.

At least for this milestone, the library function prototypes are being manually added to the symbol table when the parser is initialized. This requires a great deal of repetitive code into which typographical errors can easily creep. The ability to print out the symbol tables helped greatly with debugging these errors.

The parser's print tree routine was extended so that it would print out the types applied by the type checker, as well as printing out the Gforth library word name instead of the input name token for the head of a list node. However, all of the existing tree printing functionality was preserved so that it is possible to print out a tree that has yet to be type checked or has been only partially type checked. This, as with the symbol table printing routines, was extremely helpful in debugging the type checker and ensuring correct operation.

A.3.3 Testing Concerns

By using the print routines for both the trees and the symbol tables I was able to make sure that the data stored in various data structures was correct. I have not, as of yet, uncovered any potential pitfalls for future milestones except for the present implementation of the library prototypes. Since I have already begun considering the next milestone, I am already working on the design of a system to read the prototypes from a file at startup, eliminating the huge source of potential error in the code that manually inserts entries into the symbol tables for library functions. The keyword

constants, such as true and false, will continue to be manually inserted by the code, since they are actually part of the language more than part of the Gforth library.

A.4 MILESTONE V

Performing the actual translation to Gforth, at this step, was quite easy. As discussed in class, for the case of constants-only SOL input, all that is necessary is outputting into postfix. It was necessary for me to take special care when handling strings, however, as Forth only guarantees that a single immediate string, as specified by sör s stand and", will be held in memory at a time. To solve this issue, I used string utility functions that I had already written for use in manipulating strings in my library, strcpy and createString. To output a string constant, the translator first outputs a createString word prefixed with the correct length, then the constant string is output, using the C-style escape format, and then a strcpy word is output, copying the temporary string into heap storage. Of course, a great deal more work than this will be necessary to implement flow control structure and scope-producing statements that go along with functions and variables.

A.4.1 Implementing a Library in Gforth

Implementing the arithmetic and comparison operators in Gforth is extremely easy in general, since it is of course necessary for Gforth itself to support the operations. Doing the type conversion and argument reordering is uniform across each input type format of the operator. Because Gforth does not have an integer exponentiation or floating-point modulus operator, it was necessary to implement those myself. The integer exponentiation word was already implemented for the first milestone, so I used that directly, but for the floating-point modulus it was necessary to implement it anew. The string operations took thought and a great deal of planning the stack effect. Initially I created the utility functions createString and strcpy. Using these I created the string concatenation operator + which first creates a new empty string with the length of the sum of the two input strings' lengths. It next uses strcpy to copy the first string into the first part of the new string and then uses strcpy to copy the second string to the space following the first string. The strings are stored in the preferred Gforth format of a (c-addr u) pair, so no null termination is necessary. To implement the string comparison operations, I implemented the C library's strcmp directly, so it returns -1 if string2 is "greater" than string1, 1 if string2 is "less" than string1, and 0 if they are equal. Using this function, implementing each of the comparison types was trivial. The only two remaining

string functions are insert and charat. Both of them work by addressing a specific character in a string and returning the string back. The charat word simply reads out the contents of the memory location while insert replaces the character at the location with the given character. Since we decided in class to leave off the file I/O words, I did not implement them.

A.5 REFLECTION

Implementing the tree to Gforth translation did not take any particular effort on my part, but it was necessary to take care when designing the stack effect for the more complicated library words. In particular the string functions, since it required pointer arithmetic without the use of registers. The only safe way I discerned it possible to design these functions was to determine the "target" stack state that I needed to perform an operation (like strcpy or +), and then try to determine the least number of operations necessary to reach that state. This worked out very well for me, and made serious debugging unnecessary. In going along with this design methodology, I included step-by-step stack effect descriptions at key points, making it possible for me to understand what is occurring at a later stage. Commenting like that helps me later on once I have forgotten why I coded something a certain way.

A.6 MILESTONE VI

In approaching the front-end side of user-defined variables, I used the approach of separating each scope into its own symbol table. For my data structures, this was easiest, since it did not require any deletions. In determining the type for a variable, the user can either explicitly assign it using the `[: <name> <type>]` format or the type checker will automatically determine it from the type of the initial value expression. Either way, a new entry is placed into the appropriate symbol table and one of two stack offset counters is adjusted to reflect the new local variable. For this milestone, each scope was treated as a stack frame, so the only method that needs to perform the stack simulation is the instance of the type checker function which is type checking the let block. Begin-end blocks also create their own "scope," but it is not possible to declare any variables in it, because it is then necessary to use a let construct to actually do the declaring. For this reason, only let constructs do any scope or stack frame creation and deletion.

A.6.1 Stack Frame and Gforth

To develop an operational model for dealing with stack frames in Gforth, I first sat down with Gforth and the various stack pointer words such as `sp@`, `sp!`, `fp@`, etc. After studying their operation, I decided to store both the argument and floating point stack points together in a double word variable called `_#CURFRAMEPTR`. The code I wanted to use to produce this value was `sp@ fp@`, so, because the Gforth stack grows down, the floating point frame pointer is stored in the lower half of `_#CURFRAMEPTR` while the argument stack frame pointer is stored in the upper half. Working with this design in mind, I developed routines to make stack frames, delete them without returning a value, and deleting them returning one of the SOL acceptable types as a return value.

In creating a frame, the current value of `_#CURFRAMEPTR` is pushed onto the stack, taking two cells. After this, the new stack frame pointers are stored into `_#CURFRAMEPTR`. Instead of "allocating" space on the stack for the total space needed for all variables, I simply had the initial value code for each local variable produce its value and place it on the appropriate stack. In returning from a stack frame, the previous stack frame pointers are pushed onto the top of the stack and then written directly to the stack pointers using `sp!` and `fp!`. Though an extremely unlikely situation to occur in a functional language, it prevents the possibility of accidentally having too many elements left on the stacks, causing the machine to go into an unknown state upon leaving a stack frame. If a value needs to be returned, it is copied into a global holding variable, the stack frame is destroyed, and then it is pushed back onto the appropriate stack. Though this solution may not be as refined as it possibly could be, it does the job in less running time and coding time.

A.6.2 Variable Assignment

The other new issue in this milestone is variable assignment. To simplify the process on the type checker side, a function prototype was added to the library for each assignment operator. This function looks like a normal function to the type checker and thus requires no changes whatsoever to work with. The translator, however, must recognize that the destination needs to be passed by reference instead of by value. Correspondingly, the actual Gforth words that perform the assignment expect the first argument to be a pointer to the destination rather than the variable's value. This does put the entire job of handling variable assignment onto the shoulders of the translator, but it simplifies working with the code because there is only one component to worry about.

A.6.3 Reflection

Working with scopes inside of my compiler was not difficult, but it took a great deal of time and effort to work through and debug the stack frames in Gforth. In the x86 architecture, all arguments are passed on the stack, as with Gforth, but there is only one stack. The most complicated part, for me, of working with stack frames in Gforth is trying to keep both stacks in the appropriate state. Since stack frame pointers also point to stack frame pointers, I had to be careful about how many pointers I dereferenced. Going one stack frame too far, especially when testing in a shallow set of stack frames, easily results in going below the bottom of the stack and getting hate mail from Gforth. Once I worked out each set of functions, I then just had to create bizarre nested functional code to try to break my compiler. I made sure to test cascaded lets, the situation where a child let hides the existence of a local variable in a higher scope, and also assignments. It is truly interesting to have written a compiler that does even this much.

A.7 MILESTONE VII

This was quite a challenge to get working correctly. I started out, as with local variables, working with Gforth, figuring out how I should implement it, and then writing words to encapsulate that implementation. I then set about adapting the type checker to correctly type check functions. Previously, for milestone VI, I had allowed the user to implicitly type variables. This became a problem in this milestone, because it is difficult to tell if the user is trying to declare a new variable or call a function with a single argument. Because of this, I mandated the explicit typing syntax for both variables and functions. This simplified the code, because there were far fewer possible variations that had to be dealt with. Additionally, I allowed the variable and function declarations and expressions inside a local scope let to be ordered any way the user desires. This too reduced the code required to perform the type checking and made it easier to integrate function definitions into my compiler.

In translating functions, I discovered for myself that Gforth does not allow for variable or colon definition of new words inside a colon definition, so it was necessary for me to go back and "float" prototypes of locally defined functions into the outermost parent function. With this in hand, I had the translator output a Gforth "defer" for each function, as well as outputting the function frame pointer variable declaration. This, combined with name mangling, provides an effective method for dealing with locally defined functions in SOL.

In Gforth, I created a library word for padding the current stack frame, so as to make space for the local variables, and altered the destroyFrame

words so that they took the extra offset needed to clear off the arguments on the two stacks. The procedure is then to first create the stack frame, which saves and updates the previous global stack frame pointer, then to save and update the function frame pointer, which is used by other functions to get at local variables not belonging to themselves. The main issue with my solution is that it took a fair amount of work to determine what to start the offsets at and how much to offset by to get at the arguments.

A.7.1 Reflection on This Milestone

This milestone took a great deal of testing and debugging to complete. Most of the test cases that I created uncovered a problem, which then had to be fixed. Misalignment of offsets and the improper manipulation of variables were the main issues. My schema for where local variables are and how they are accessed changed significantly from milestone VI, but some of my code was not properly updated to reflect that change. Some way of associating code with its design element would have helped me to resolve the issues sooner. The largest overall issue for me was the fact that there were three languages involved when doing any testing, each with a complete set of semantics. On previous milestones, the input complexity was very limited, also limiting the output complexity which made it easier to figure out what was wrong at some point. Here, the issue could be, if the compiler is throwing an error, that either the input file or the compiler has an error. That kind of error is somewhat easier to deal with by looking at a very familiar programming language, C, and analyzing its operation to figure out what is wrong. Because of this, it has been easy for me to fix problems without directly encountering them, merely anticipating them when I am going through the code looking for a separate error. If the output program does not work, then it could be any of the three, and it takes me a great deal of time to determine the exact problem. Gforth does not give line numbers for errors inside compiled words, so it is necessary to insert print statements to determine where it is exactly. Then it is necessary to examine the stack before and after certain events, to make sure that the correct number and type of values are being generated, and finally I frequently have to look at the semantics for my library words to make sure that what it should be doing corresponds to what I expect it to be doing.

A.8 REFLECTION ON THIS COURSE

I find compilers to be a very interesting subject. This course was very challenging and time intensive. The only reason I completed through the final milestone was that I always worked to stay on top of all of the milestones, which did become progressively harder throughout the semester. It seems

that this semester has, for everyone I know both in and out of this course, been the busiest yet, making it extremely difficult to keep on top of the workload for this course especially. It would have been extremely helpful if we had gotten an earlier start on the earlier milestones and finished them sooner, giving more time for the later milestones, which truly require it. It might also have been useful to have gone through all the components of the compiler sooner and worked on designing the complete solution, rather than piecemeal. Doing as it was done did, however, require a great deal of thought at each point to figure out how to go from where I was to where I needed to be. The required course book was not very useful through this process. I found other books at the library that actually described the components required to implement a compiler which were more helpful in designing my milestones. The project book would be more useful if it had explicit examples of the SOL code in use and if the milestone descriptions actually described what was necessary for completion of the milestones.