CIS*4650 - Compilers Milestone 1

Boris Skurikhin, Brayden Klemens

March 2022

Techniques:

- We began writing our parser bottom up. We wrote all “final” productions first, and the more complex productions later in the process. This technique allowed us to carefully plan our grammar structure and helped us avoid a wall of errors.
- Removing ambiguities in grammar by way of leveraging the precedence declarations in the cup file. Instead of manually adjusting the grammar, we found this technique to be extremely useful because it removed ambiguity and made our grammar more readable.

```
precedence left PLUS, MINUS;
precedence left TIMES, DIVIDE;
```

- We found it useful to have a large number of Absyn classes to better represent the tree (make it more readable). For example, when working with an expression like “arr[3] = 5”, initially, our syntax tree output looked somewhat confusing though it was correct. We created a special symbol for this scenario that would help us recognize when something is being assigned to an array at some index.

Design Process:

- The design process started with the creation of the .flex. Our flex file contains all the possible regexes used for scanning the C- language. We also took care of C-style comments, and left a sink state for any possible errors. We then defined

all the terminals (used by the flex file) along with all the non-terminal states. We followed the grammar specified in Checkpoint 1 document while keeping tabs on precedence. We built embedded code as we built out productions to ensure that everything works together, and to avoid running into problems all at once in the end. Lastly, we implemented error recovery, handling the most obvious errors first, then looking into obscure cases last.

Lessons Gained in the implementation process:

- Testing code often; this might sound generic, but it helped us a lot. There were a few occasions where we got too confident and wrote too much code that had to all be deleted.
- Refrain from passing null values into our abstract syntax tree structures, as it eventually leads to null pointer exceptions when attempting to print the syntax tree. By the end of the assignment we had to replace nulls with object stubs that were most relevant in the context.

Assumptions:

- For the sake of the test files, we assume that all of the test files (1-5.cm) should end with the .cm extension.
- We assume that only 1 .cm source code file will be passed into the compilation, and its name/path does not start with the symbol “-”

- As per the project outline, our program requires -a flag to run the parser and output the syntax tree to a file. If the “-a” flag is not present, the execution of the program will halt right away.
- Assume that the errors should only be written to stderr, error messages will not appear in the syntax tree file.
- Any error recovery in the syntax tree is displayed with an ‘ERROR’ identifier.

Limitations:

- Our compiler does not have error recovery for top level declarations such as an invalid function declaration, or statement declarations. For example if we were to define a function with an invalid type such as ‘String function(param, param)’ our compiler does not have error recovery for this scenario. Thus, the compiler will produce a syntax error, and will fail to continue the parsing for all top level declarations.

Improvements:

- As mentioned in our limitations, our compiler cannot identify improper function declarations, this would be a major improvement for our compiler because it would allow us to continue parsing and identify other errors that occur inside the top level.
- Extending the grammar in the future, and adding more error handling, would allow for more complex programs. Such as handling negative numbers.

Error Recovery We Handled:

- Empty files are not allowed.
- Invalid variable declarations such as: improper names, array sizes, and types.
- Invalid parameters in functions such as declaring a value.
- Invalid statements (if, else, loop, return). This occurs when the expressions inside of the statements are invalid. These include things like improper variable names and bad comparisons.
- Function calls cannot have invalid function call parameters.
- When assigning array values, the array cannot have an invalid index.

Contributions:

- As a team we worked over discord to tackle most of the problems as a team, however we were able to split up the code once we outlined what needed to be completed for each step of the implementation process. VsCode live share was a very useful tool in our pair programming process, which allowed us to update the code base in real time. This prevented a lot of confusion when pushing changes to our CUP file. We worked on the compiler together, while on a live call. This type of teamwork allowed us to discuss trade offs and ideas in real time and be up-to speed with the latest code.

Boris:

- Parser, Jflex, Grammar rules, Error recovery, Abstract syntax tree structures

Brayden:

- Grammar rules, Error recovery, Abstract syntax tree structures