CIS*4650 (Winter 2021) Compilers

Implementation Project - Checkpoint One

Due time: March 8, 2021 by 11:59 pm.

Functionality:

For Checkpoint One, the following functions should be implemented:

- Scanner that generates a sequence of relevant tokens
- Parser that generates and shows an abstract syntax tree
- Error Recovery that reports errors and continues the parsing process

A sample parser for the Tiny language is provided along with this description in our CourseLink account: "SampleParser.tgz". You are allowed to use it as a starting point for your implementation, as long as you acknowledge it in the README file of your submission.

For the C- language, the CFG for its syntax is already given in the related document "C-Specification". However, it uses the layering technique to remove the ambiguities for the relational and arithmetic expressions. Since the tools like CUP allow you to specify the directives for the precedence orders and associativities of these operations, you should simplify the given grammar to take advantage of such supports. Please refer to the lecture notes or the manual for CUP for more information.

Your program should detect and report all kinds of lexical and syntactic errors from the input. You should handle error recovery in the most reasonable way possible, always attempting to recover from an error and parse the entire program so that you may detect multiple errors in the process. You will find some kinds of errors are easier to handle than others in a graceful manner. You should try your best under the circumstances, and certainly handle cases that are obvious or easily detected. Obviously, it is not realistic for you to handle all possible error conditions (at least not within a semester time) but do what is reasonable and realistic in the given timeframe.

Execution and Output

Your compiler should be able to process any C- programs to the point of building an abstract syntax tree and show the hierarchical structure clearly in the output. In other words, you are required to implement the `-a` command line option in the compiler project. Please refer to the lecture notes on "6-Syntax Trees" for the recommended syntax tree structures for the C-language. In addition, syntax errors should be reported to stderr in a consistent and meaningful way (e.g., indicating the line and column numbers along with a brief explanation of the related error). You may find it useful to check how other compilers report syntax errors.

Documentation

Your submission should include a project report describing what has been done for this checkpoint, outlining the related techniques and the design process, summarizing any lessons gained in the implementation process, discussing any assumptions and limitations as well as

possible improvements. If you work in a group of two, you should also state the contributions of each member. This does not mean you cut and paste from the textbook or lecture notes. Instead, we are looking for insights into your design and implementation process, as well as an assessment of the work produced by each member if relevant. The document should be reasonably detailed, about four double-spaced pages (12pt font) or equivalent, and organized with suitable title and headings. Some marks will be given to the organization and content of this document in the marking scheme.

Testing Environment and Test Programs

You need to verify your implementation on a Linux server at linux.socs.uoguelph.ca, since that will be the environment for evaluating your demos. In addition, we encourage you to follow these incremental steps to build your implementation: (1) build a scanner for the C-language; (2) implement a parser that only contains the grammar rules (i.e., no embedded code for this step) and get it connected to your scanner; (3) add the embedded code for the grammar rules so that you can produce the syntax trees; and (4) incorporate error recovery so that you may catch multiple syntax errors during the parsing process.

In addition, each submission should include five *C-* programs, which can be named `[12345].cm`. Program `1.cm` should compile without errors, but `2.cm` through `4.cm` should exhibit various lexical and syntactic errors (but no more than 3 per program and each should show <u>different</u> aspects of your program). For `5.cm`, anything goes and there is no limit to the number and types of errors within it. All test files should have a comment header clearly describing the errors it contains, and what aspect(s) of the errors that your compiler is testing against.

Makefile

You are responsible for providing a Makefile to compile your program. Typing "make" should result in the compilation of all required components to produce an executable program: *CM*. Typing “make clean” should remove all generated files so that you can rebuild your program with “make”. You should ensure that all required dependencies are specified correctly.

Deliverables

(1) Submission to the related drop box on CourseLink:
- All source code required to compile and execute your “*CM*” compiler
- Makefile and all the required test files
- The README file for build-and-test instructions
- A project report as described in the “Documentation” section above
- Tar and gzip all the files above and submit it on CourseLink by the due time.

(2) Demo: You should schedule a brief meeting of about 15 minutes with the instructor or one of the TA’s so that we can evaluate your demo on March 9, 2021. A link to the shared document will be emailed to you several days ahead so that you can sign up on an available time slot for your demo. The meeting will allow you to demonstrate your implementation and get feedback from us as you embark on the Checkpoint Two of your compiler project.