CIS*4650 (Winter 2018) Compilers

Implementation Project - Checkpoint One

Due time: March 5, 2018 by 11:59 pm.

Functionality:

For the first checkpoint, the following functions must be implemented:

- Scanner
- Parser (to the point of building an abstract syntax tree)
- Error Recovery

Two sample parsers for the Tiny language are provided for your reference: "c_tiny.tgz" for C implementation and "java_tiny.tgz" for Java implementation. You are allowed to use them as a starting point for your implementation, but you need to 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 Yacc/CUP allow you to specify the directives for the precedence orders and associativities of these operations, you should simplify the grammar to take advantage of such supports. Please refer to the lecture notes or the reference for Lex/Yacc for more information.

Your program should detect and report all lexical and syntactic errors from the input. You should handle errors in the most reasonable way possible, always attempting to recover from an error and parse the entire program, without segmentation faults or core dumps. Note that it is possible that the parsing process will reach a point where recovery is not possible. In such cases, your program should terminate rather than dumping the core. Whenever possible, your program should attempt to recover from a parse error and continue the parsing of the source code, possibly detecting multiple syntax errors in the process.

You will find some kinds of errors are easier to handle in a graceful way than others. You should do your best under the circumstances, and certainly handle cases that are obvious or easily detected. Of course, it is not realistic for you to handle all possible error conditions (at least not within one semester time), but do what is reasonable and realistic in the given time period.

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 meaningfully in the output. In other words, you are responsible to implement the `-a` command line option in the compiler project. Please refer to the lecture notes on "6-Syntax Trees" in our CourseLink account for the recommended syntax tree structures for the C- language.

Syntax errors detected by your parser should be reported to stderr in a consistent and relatively meaningful way (e.g., indicating the line number and providing a brief explanation of the related error). You will find it useful to check how other compilers report syntax errors during parsing.

Documentation

Your submission should include a document 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 as 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 group member where appropriate.

The document should be reasonably detailed, about four double-spaced pages (12pt font) or equivalent, and should be organized with suitable title and headings. Some marks will be given to the organization and presentation of this document.

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) Documentation (hard-copy) should be submitted at the time of your demonstration.

(2) Electronic submission:

- 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
- Tar and gzip all the files above and submit it on CourseLink by the due time.

(3) Demo: You should schedule a brief meeting of about 15 minutes with the instructor or the TA on March 6 so that we can evaluate your demos. A shared document will be posted several days ahead as a news item on CourseLink so that you can sign up on the available time slots for your demos. This meetings will allow you to demonstrate your implementations and get feedback from us as you embark on the next stage of your compiler projects.