CIS*4650 (Winter 2016) Compilers

Implementation Project - Checkpoint One

Due time: March 7, 2016 by 11:59 pm.

Functionality:

For the first checkpoint, the following functions must be implemented in your project:

- 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 and "java_tiny.tgz" for Java. You are encouraged to download the one of your choice and play with it early so that you can get familiar with the recommended structure for the implementation.

For the C- language, the CFG for its syntax is already given in the related document "CMinus". 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 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 be able to detect and report errors in syntax. 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 attempt to 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 the scope of one semester course), but do what is reasonable and realistic in the time you have available.

Execution and Output

Your compiler should be able to process any possible C- programs to the point of building an abstract syntax tree to represent the phrase structure of the source code (i.e. you are not required to produce assembly language output or any intermediate data structure beyond the abstract syntax tree at this point).

Syntax errors detected by your parser should be reported to stderr in a consistent and relatively meaningful way. You will find it useful to consider how other compilers you may have used report syntax errors during parsing.

For this checkpoint, you are responsible for implementing the `-a` command line option (i.e. output a representation for the abstract syntax tree).

Documentation

Your submission should include a document describing what you or your group has done for this checkpoint as a whole, along with the contribution of each group member if relevant. 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 group member assessment should be brief, but specific about what each member did, and how it fit in with the group effort.

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

Test Programs

Each submission should include five original *C-* language programs. These test programs may be used to evaluate your own implementation. It should go without saying that you should be submitting programs that your own compiler works with.

The test files 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 in order to produce an executable program: *cm*. Typing “make clean” should remove all generated files so that you can type “make” again to rebuild your program. You should ensure that all required dependencies are specified correctly in your Makefile.

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
- The required README file for build-and-test instructions.
- Tar all the files above and submit it to the course website by the due time.

(3) Demo: You should schedule a brief meeting of about 15 minutes with the instructor for a demo on March 8. This meeting will allow you to demonstrate your implementation and get feedback as you embark on the next stage of your compiler construction.