CIS*4650 (Winter 2024) Compilers

Implementation Project - Checkpoint Two

Due time: Mar. 18, 2024, by 11:59 pm.

Functionality:

For the second checkpoint, the following functionality should be added to the compiler project:

- Symbol Table
- Type Checking

~~After Checkpoint One, your program should generate an abstract syntax tree if the input is valid, otherwise, it should detect and report syntactic errors.~~ With the implementation of Checkpoint Two, your program can now detect and report semantic errors such as mismatched types in expressions, and undeclared/redefined identifiers. This can be done by traversing the abstract syntax tree generated in Checkpoint One in the post-order, which should be implemented in another "visitor" class such as "SemanticAnalyzer.java" so that all the related methods can be isolated in one class. Once again, you should handle errors in the most reasonable way possible, always attempting to recover from an error so that we can process the entire program and capture as many semantic errors as possible. Please refer to the lecture notes on "8-Type Checking" for the related concepts and implementation tips.

Execution and Output

Your compiler should be able to handle any possible C- programs to the point of building abstract syntax trees, maintaining the symbol tables, and performing type checking for the semantic analysis.

Errors detected by your program should be reported to stderr in a consistent and relatively meaningful way (e.g., indicating the line and column numbers and providing a brief explanation of the related error). You will find it useful to check how other compilers report compilation errors during parsing and semantic analysis.

For this checkpoint, you are also responsible for implementing the -s command-line option for displaying symbol tables. Note that symbol tables change as you go through different scopes, and for each scope, we can only gather all of its symbol definitions at the end. Accordingly, you can write messages of when you enter and leave a scope and display all symbols for each scope just before you leave the scope. In addition, the scopes are fully nested, and thus, you should indent the information for each scope according to the nesting levels (please see Slide 12 in the notes on "8-TypeChecking" for an example).

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 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 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.

Test Environment and Programs

You should verify your implementation on the Linux server at linux.socs.uoguelph.ca, since that will be the environment for evaluating your demos. For incremental development, you can implement the symbol table first and then perform type-checking along with error recovery. In addition, each submission should include five *C-* programs, which can be named `[12345].cm`. Program 1.cm should compile without errors, while 2.cm through 4.cm should exhibit different kinds of semantic errors (but no more than 3 per program). For 5.cm, anything goes and there is no limit to the number and types of errors in 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 type “make” again to rebuild your program. 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 one of the TA’s so that we can evaluate your demo on March 19, 2024. A link to the shared spreadsheet 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 Three of your compiler project.