CS 4620
assigned: 09/23/2022
due: 10/21/2022 at 5pm

12 points

Course Project Deliverable 2

The course project provides a context for you to study concepts related to compilation more deeply. This is the second of 5 project deliverables:

Deliverable 1 Extend the parser to support SIP

Deliverable 2 Extend the front end to support SIP

Deliverable 3 Extend the semantic analyses to support SIP

Deliverable 4 Extend the code generator to support SIP

Deliverable 5 Extend the optimizer for SIP

The project is solved in teams of two, unless you want to do it by yourself.

Deliverable 2 : Problem Statement

You extended the parser with SIP support in deliverable 1. In this deliverable, you extend the rest of the compiler frontend with SIP support. More specifically, you will extend the:

AST adding new subtypes of `ASTNode` in the `tipc/src/frontend/ast/treetypes` directory;

AST Visitor adding new methods to the visitor to support the new subtypes;

AST Builder adding new methods to build AST nodes from SIP parse trees; and

Pretty Printer adding new methods to print a textual version of the AST.

As with deliverable 1, you must not introduce any errors in processing TIP programs, i.e., programs written without any of the SIP extensions, unless you find illegal TIP programs become legal SIP programs due to the extensions.

You are to provide CATCH2 unit tests that thoroughly exercises the functionality of your frontend. You can build on the existing unit tests available for the frontend. In this deliverable, you are expected to provide testing that covers every line of the new functionality you implement. You can determine this by running the `tipc/bin/gencov.sh` script and then loading the `tipc/coverage.out/index.html` file in a browser and exploring your coverage.

Deliverable 2 : Accessing the Project, Solution Expectations and Grading

Work on this deliverable is a continuation of your prior deliverable. Just keep working in the same repository that your team has been using.

Course Project Deliverable 2

12 points

You are expected to extend and modify, as necessary, the implementation of the `tipc` frontend, to extend frontend unit test suites with new test files beginning with the `SIP` character sequence, and to provide a brief description of your work. The description of your work should in a file `SOLUTION-D2.md` that you add to the root directory of your project. You should briefly describe any tricky aspects of your solution, any design alternatives you considered and how you decided among them, and your approach to testing and achieving high test coverage.

We will access the latest project commit prior to 5pm Eastern Time on Oct. 21, 2022 to assess your solution. You may continue your work on the project after that time, but it will not be considered for grading this deliverable. Grading will be based on the following rubric:

Quality of Solution (7pts) The completeness and correctness of your solution when considering both the SIP extensions and the preservation of TIP support.

Quality of Testing (4pts) The completeness of testing of your solution.

Quality of Description (1pt) The completeness and quality of the description of your solution.

Deliverable 2 : Scope, Strategy, and Suggestions

To give you a sense of the scope of work for deliverable 2, not including our test code, our solution is comprised of:

- 11 new header files and 229 new lines of header file source code
- 11 new cpp files and 315 new lines of cpp source code

This is a pretty substantial project because you need to understand the organization of the frontend, e.g., the `ASTNode` type hierarchy, the way that the `ASTVisitor` is implemented and the support required by new instances of `ASTNode` subtypes, the idioms for constructing `std::unique_ptr` instances for new AST nodes, the code structures used for collecting data during a post-order traversal in the `PrettyPrinter`, etc.

We suggest an incremental strategy where you first focus on extending the `ASTNode` type hierarchy with subtypes to represent the new SIP constructs. You would be well-served to base your work for a given SIP feature on closely related TIP features, e.g., using the AST support for TIP features as a model for your work on this deliverable.

Once you have a *code complete* AST then you would be wise to immediately move to developing unit tests to demonstrate to yourself that it works as intended. Note that there are no existing unit tests for you to use as a model for testing the AST in isolation. You will need to develop an approach to unit testing that, for example, constructs instances of the `ASTNode` subtypes and then uses the API of the subtypes to access the values. You can assume that the TIP support is solid and just focus on your new SIP support.

With the AST for SIP in place you should move on to the `ASTBuilder`. In developing the unit tests above you will be forced to understand how to construct instances of `ASTNode` subtypes. This will serve you well in this step, since that's what the AST builder does. Here again model your work for a SIP feature on existing support for building related TIP features. This will be especially important as you work to understand move semantics and how they relate to the use of `std::unique_ptr`; the TIP features provide a model for you to learn from.

You will need to familiarize yourself with the names of functions and member data from the ANTLR4-generated `TIPParser` context data structures to be able to access the parse tree information; recall that your changes for deliverable 1 will introduce new parse tree nodes with new names. Testing the AST builder extended for SIP can follow the model of the `ASTPrinterTest`.

The last element of the deliverable is extending the `ASTVisitor` and then the `PrettyPrinter`, which uses it. Once you are to this point you will be very familiar with the AST and this should not be all that challenging, but only because you climbed a big learning curve above.

The dependences between the components of this deliverable make certain divisions of work undesirable. For example, if one team member worked on the AST extension and another on AST building, the second would be stuck waiting for the first to finish.

It is possible, however to divide the deliverable by feature. For example, one team member could implement support for Boolean types and push that all the way through the sequence of steps outlined above while another did the same for For loops; then you could merge. The risk with this approach is that you might make different decisions about certain internal details in your implementation that clash or that lead to redundancy, e.g., if you both need the same temporary variable and name them differently.

A division of work that we think makes sense is to work through the major steps of the deliverable in sequence dividing the work for each. For example, if you both work together on the first `ASTNode` subtype implementation and develop a shared understanding, then you agree on what other subtypes are needed then you can divide those between yourselves and work in parallel.