Generating ILOC from DEMO

Programming Assignment 2 — DUE DATE: 11:59 PM on
COMP 506, Spring 2019 — Monday, March 19, 2019

1 Introduction

This project will give you experience generating code from the parser that you built in the first programming assignment.[1] You will augment the code that you wrote for assignment one to generate ILOC code for each of the constructs in DEMO. This hands-on experience should improve your understanding of both how compilers operate and how language constructs are implemented. The fundamental goal of this exercise is to help you understand the actual cost of the various abstractions provided in a programming language.

From a practical perspective, you must (1) familiarize yourself with ILOC, the target language for the assignment; (2) add to your parser a mechanism that performs storage mapping, after parsing the declarations and before parsing the executable statements; and (3) add semantic actions to your parser that emit ILOC code to implement the executable DEMO statements. Of course, you will need to build infrastructure to support the code that you add to the parser.

All documents for this lab, along with the executable ILOC simulator and a reference implementation, are available from the course web site.

2 Project Summary

In this assignment, your task is to:

1. extend the parser that you built in the first assignment so that it emits ILOC code to implement any correct DEMO program, and
2. test the resulting compiler using the ILOC simulator provided for that purpose.

If the DEMO program presented as input is syntactically correct, your compiler should produce correct ILOC that faithfully implements the input program. It should write that ILOC output to a file with the same "basename" as the input program—that is, if the input is `test_program.demo`, the output should be in `test_program.i`.

Your product for this project is a working compiler for DEMO that produces correct ILOC code. You will submit a `tar` file that contains the source code, a `makefile` and a `Readme` file. The makefile should compile and link your lab. It should rename the executable from `a.out` to `demo`. The `Readme` file should explain how to build and execute your compiler.

Your compiler should be invoked with the same syntax as your parser from the first assignment. The command-line syntax should be

```
demo [-h] [filename]
```

where items in square brackets ([and]) are optional. The optional `-h` flag should print the command line syntax to `stdout` and quit. The optional `filename` argument specifies a file to be parsed. If `filename` is missing, your parser should read from `stdin`. If you add other command-line flags, the `-h` option should describe their effects.

[1] If you prefer not to use your own parser, contact the instructor and he will provide one.

3 Suggested Approach

1. Find and read all the documents that are relevant to the project. That should include, at a minimum, this document, the Demo language specification, the ILOC language and simulator document, and the relevant lecture notes. All should be on the course web site.
2. Extend your parser so that it performs storage mapping—that is, it should assign a location (either a register or an address in memory) to each declared variable in the Demo program.

   Undoubtedly, this task will require that you add a symbol table where the productions for declarations (in Demo) can record the information provided by the declarations (for example, the variable's type and its dimensions if it is an array).
3. Incrementally add actions to your parser (and support code) to emit ILOC operations that implement various parts of the grammar. Early implementation of Demo's `write` statement for scalar variables, may help in testing more complex constructs, such as array addresses and control structures.

4 Advice

As with the parser project, incremental development and testing is probably easer than trying to enter the code for the entire project and then test it. Start with small parts of the grammar, such as the `read` and `write` statements. Branch out to handle assignments to scalar variables, maybe from another variable or a literal constant. Test thoroughly by reading the ILOC code, and by running the ILOC code with the simulator.

The directory `~comp506/students/demo/lab2` on CLEAR contains small demo programs that may be useful in developing, exercising, and debugging your code.

Mid-Production Actions While `bison` provides a mechanism to insert actions in the middle of a production, the naming convention in those mid-production actions is a bit confusing. The mid-production actions must use the `bison` macro-names from the full production, while not referencing items to the right of the action. To return a result from the mid-production action, the compiler-writer can assign to $$ (and pay attention to the sequencing of those actions) or to some other item to the left of the action. Often, the values in all those locations are useful.

Splitting the production manually, by adding a new non-terminal symbol and an epsilon production has the added benefit of creating a new symbol in the original production that can hold a return value from the (mid-production) action.

Another source of confusion from mid-production actions is the fact that they can introduce new shift-reduce or reduce-reduce conflicts. Splitting the production manually at least makes those conflicts more obvious.

ILOC ILOC is described in a separate document. The ILOC simulator documentation presents the full range of ILOC operations, along with directions for how to use the ILOC simulator to execute an ILOC program.

You should read the ILOC simulator document to gain an understanding of ILOC *before* you begin coding the semantic actions into your parser. You can use the lab2 reference implementation to see the code that it generates for specific constructs or whole programs. The reference implementation is available on CLEAR in the directory `~comp506/students/lab2`.

The simulator's trace facility (`-t` command-line flag) produces a detailed trace of the operations that the simulator executes and the values that it computes into registers. From the trace, you can follow the execution of an ILOC program; in practice, the trace facility is one of the most important debugging tools that you will have for this assignment.

Debugging In developing the reference implementation, we found it convenient to produce a storage layout report so that we understood where in memory variables had been allocated. That report appears in the `.sl` file. We also implemented a flag that allocated all variables to memory, the `-m` flag. These two additons made it much easier to test the code and to understand the results.