COL728 : Compiler Design : Labs (programming assignments)

All lab assignments are to be done individually.

Lab 1 : Studying the C parser

Due date

Part 1: 18th August 2022, Thursday
Part 2: 2nd September 2022, Friday

Weight

5+5 = 10 marks

Lab instructions

1. Clone the Git repository: https://github.com/iitd-plos/cc
   ```
   $ git clone https://github.com/iitd-plos/cc
   ```
2. In `cc` directory, type `make` to build the `cc` program
3. The `cc` program checks if an input C program is correctly formed (as per the C grammar). The C program should not have any pre-processor directives, and is assumed to have been pre-processed already. Try compiling the `hello_world` program:
   ```
   $ ./cc examples/hello_world.c
   ```
   Try changing the `hello_world.c` program to valid and invalid C programs, and check if the compilation succeeds or reports a syntax error appropriately.
4. ***[Part 1]:* Study the generated lexer in file `c.lex.cpp`, and write a short report (1-5 pages) on the logic of the code produced through the lexical analyser generator**. In particular, we are interested in the logic of the function represented by YY_DECL (representing the `yylex()` function). Keep your report as brief as possible, but do not miss out any interesting fact about the generated code or logic. Your report should reflect your understanding of the logic of the generated lexical analyzer.
   - *Hint*: Track the `yy_act` variable.
5. ***[Part 2]:* Study the generated parser in file `c.tab.cpp`, and write a short report (1-5 pages) on the logic of the code produced through the parser generator**. In particular, we are interested in the logic of the `yyparse()` function. Keep your report as brief as possible, but do not miss out any interesting fact about the generated code or logic. Your report should reflect your understanding of the logic of the generated parser.
   - *Hint*: Track calls to the `yylex()` function for reading the next token through the lexical analyzer.
   - *Hint*: Look at the `yyreduce` label, which is reached during reduction of a rule.
6. ***[Optional assignment]:* Study the hand-coded parser of initial version of Tiny C Compiler**. Comment on what parsing algorithm is being used here. On what basis do you make your conclusion?

Submission Instructions

Prepare your report in PDF format and submit using Moodle.

Lab 2 : Code Generation

Due date

Part 1: 1st October 2022, Saturday
Part 2,3,4: 31st October 2022, Monday

Weight

6 + 14 = 20 marks

Submission Instructions

1. Cleanup the `cc` directory, so that it only contains the source files.
2. Compress this directory to EntryNumber.zip file and submit it on course page on moodle.

Lab Instructions

This assigment has multiple parts, each part depends on previous part. **General instructions:**

- The goal is to modify the `cc` program to emit (unoptimized) LLVM bitcode for the input C program.
- You will start with supporting the most basic C features (e.g., those that can compile the `hello_world.c` program), and then incrementally add more C features to the LLVM code generator.
- Please see the references page for some good resources on generating LLVM bitcode. The generated LLVM bitcode should run as expected using the LLVM interpreter `lli`.
- Please note that you will be evaluated on the **correctness and elegance of your design/code**, and on the **generality/extensiveness of your handling of different C constructs**.
- Further, you do not need to handle all C constructs, but we would like you to go as far as you can. The elegance of your design is likely to make your code simpler and easier to understand and debug, and is likely to allow you to make your implementation more general/extensive in the same amount of limited time.

1. ***[Part 1]:*** **Building AST**
    1. In this part you will build an AST using the provided Bison grammar specification.
    2. Switch to the `lab2` branch of `cc` (you may need to `git pull` first).
    3. Look at the provided `c.y` file and implement support for emitting AST for two C constructs: function definitions and expressions (the `function_definition`, `expression` and other dependent parts of grammar)
        - You should have a `dump_ast()` or similar method/function which should dump the string representation of the built AST.
    4. You do not need to exhaustively support all constructs but your implementation MUST at least be able to parse and build AST for the C files in the `examples` directory, namely `test1.c`.
    5. The goal of this exercise is to have the basic infrastructure ready in order to support more complex C constructs. Being general in your implementation will you help in future parts of the assignment
2. ***[Part 2]:*** **Supporting more complex constructs**
    1. In this part we will enough C constructs in order to get the `hello_world.c` example (and a few more) working.
    2. Utilizing the infrastructure developed in previous part, add support for variable declarations, branching and control-flow statements, namely `if, while, return, goto` etc.
    3. Handling variables declaration would require tracking scopes and declarations using a symbol table.
    4. Your implementation must be able to parse and build AST for `hello_world.c` and `test2.c`.
3. ***[Part 3]:*** **Emitting LLVM IR**
    1. This part will introduce you to the LLVM IR, and at the end of this assignment your implementation will emit LLVM IR for the C code it is able to parse.
    2. The emitted LLVM IR bitcode should run as expected (per the input C code) using the LLVM interpreter `lli`.
4. ***[Part 4]:*** **Local optimizations**
    1. Extend your code generator to support some local optimizations, like local constant-folding, local dead-code removal, etc.
    2. Here is an example of a trivial constant propagation optimization.
    ```
    _Bool x, y;
    ...
    x = 1 || y;
    // this last instruction can be changed (at the time of parsing) to
    x = 1;
    ```
5. ***[Optional]:*** **More C constructs**
    1. Add support for more C statements:`for, do-while, switch`. For example, try to get Duff's device working.
    2. Handle local and global arrays.
    3. Handle implicit casting and type checking.

Some hints if you are unable to start

1. You may want to start with creating a syntax tree for a simplified grammer.
2. Remove some rules and keep only small number like function declaration.
3. Define a header file in which you can define structure of nodes for AST.
4. In c.y file, add C code for returning tree of your defined type along with grammer rules. You can put C code inside braces ({}) in front of grammer rules.
5. In each class, defining structure of node for your AST, include a function which will generate llvm code for that node. You can take some help from this link for getting started for code generation.
6. The References page lists out some useful resources as well.