Mini Assignment 1 Report

Name : Vedha Moorthy S
Roll No : CS16BTECH11040

Clang-AST Structure

I compiled some programs to observe the AST generated by clang. The entire generated code is contained in something called a *Translation Unit Declaration*. Most of the code consists only of the functions and declarations present in included headers. The last 30 or so lines described my intended program in the form of a tree.

- Fibonacci Program (Appendix A) : Tree starts (neglecting the headers) at the first user defined declaration main() as an integer *Function Declaration*. The whole main() block is composed as a *Compound Statement*. It's child nodes consists of *Variable Declarations* and *Loop Statements*. The first child of the *For Statement* subtree consists of local variable declarations, conditional and incremental expressions. The conditional expression is called here with appropriate typecasting as the first node of the *Compound Statement of For*. Assignments and printf() statements form the other nodes within the appropriate subtrees. Finally, the AST reached *Return Statement* and terminates.

- Program with Function Calls (Appendix B) : Tree structure of main() is similar to previous program. For the *If Statement*, the conditional expression is the first node and the blocks of IF and ELSE form the next 2 nodes, which are entered by checking the conditional expression value. The *Function Declaration* of diff() has a similar tree structure like main(). The parameters are acknowledged at the start of the function as special *Variable Declarations*. If and Return statements here are similar to that in main().

One other I noticed is the neat indentation of trees to separate different blocks/subtrees from each other. Objects of the same subtree are encapsulated together using a series of pipe symbols (‘|’).

To summarize

- LLVM has three types of classes : declarations, statements and types.
- Declarations include Parameters and variables.
- Statements include Compound, If, For and Return.
- Types include the various types in C like Array, Pointer etc.

Clang-AST Traversals

To traverse the AST described above, LLVM has traversal methods for each type of tree nodes. One of the methods is using a Recursive AST Visitor.

- In short, it is a depth-first traversal of the entire AST.
- Its visits all nodes of AST starting from an entry point by the Traverse Declaration.
- At a node, it goes through the class hierarchy from Dynamic type to a top-tier class (the 3 described above).
- It doesn’t enter each node but rather calls another function to visit the node.
- Preorder traversal is the default but it can be overridden to postorder.
- All parts of the code are visited mostly once.

LLVM-IR

IR is a low-level programming language on a similar level as assembly. The LLVM frontend for all languages generate an intermediate code in the common language IR. the IR code is passed to a LLVM optimizer before backend conversion for specific architectures. Here, I have analyzed the ll files of a few non-trivial C programs containing functions, switches, strings, loops and structs. Code in Appendix C.

- The common items for all programs are the source file name fields, data layout format and target machine.

- The basic instructions available in IR include operators (add, sub, cmp), store, load, branch, call, return etc. (similar to assembly).
- The functions (including main) are put inside a *define* block containing their contents.
- Statements are restructured with expressions which operators aligned as functions followed by arguments. Temporary variables are added when necessary to hold intermediate values.
- Loops and switches are handled as separate blocks inside a function. The preheaders, condition and body divided into separate blocks, switching between them during iterations using branching statements. Multiple loops are labelled for differentiation.
- scanf() and printf() functions are recorded as separate *declare* statements.
- Structures are simply modelled as tuples of appropriate data types.

Error Messages

LLVM asserts can be used to find errors in code. Assert statements also take a string which can be displayed as error message, helping to identify which part of the code that failed the assertion. LLVM also has an alternative for asserts which asserts may not be clear or be cut from code, in form of the llvm_unreachable() function. Note that both of these do not abort the program when flagged.

Assembly language

C/C++ assembly language can be obtained from c/cpp files by any C/C++ compiler. The code consists of a long series of simple instructions designed to be machine friendly. Although the code is much simpler than IR, they both have a lot of similarities in structure and instructions.

- Name mangling is the representation of variable and function names into unique easily distinguishable names.
- Registers and simple variables replace the user defined names.
- This not only ensures separation of variables with similar names but also facilitates function overloading.
- It is controlled by compiler design, meaning different kinds of name mangling can be observed on different platforms.

Compiler toolchain and options

Some of the tools of LLVM are described below.

- *bugpoint* : debug optimization or code generation rounds.
- *lli* : LLVM interpreter, functioning as a Just-In-Time (JIT) compiler which executes LLVM bitcode.
- *llc* : LLVM backend compiler, translates LLVM bitcode into assembly.
- *llvm-as* : LLVM assembler converting human-readable bitcode into assembly.
- *llvm-dis* : LLVM disassembler which does the opposite, converting assembly back to bitcode.
- *llvm-link* : used to links multiple LLVM modules into a single program.

Kaleidoscope

Kaleidoscope is a very basic procedural programming language which can be used for a better understanding of LLVM's functioning. It has a single data type (64 bit floating number), can define functions, handle conditionals, basic maths along with if/then/else and for loop constructs.

Lexer

- Breaks up the inputs into tokens.
- Here, the lexer is designed as an enum structure which can identify end of file, the keywords 'def' and 'extern', identifiers and numbers. Other characters will be returned as ASCII values.
- A *gettok()* function is used for processing the input stream one character at a time, storing the last character yet to be processed at an instant.
- Whitespaces are skipped.
- A simple loop simulating a DFA is used to identify tokens. Keywords are checked first and tokenized first.

Parser

- Parsers build an AST which becomes much easier to evaluate during the later stages of compiler action.
- Kaleidoscope's AST has 2 base classes : one each for expressions and functions.

- Expression class captures the literals as instance variables. It’s subclasses include variables, binary expressions and function calls.
- The prototype Function class consists of the function name and its arguments.
- Further, details can defined later on using virtual methods.
- Recursive descent parsers can be used to create an AST.
- It consists of a number of routines, one of which acts depending on the current token. Tokens are consumed while the AST is being constructed.
- All the routines are handled by a driver program which assigns the correct routine based on tokens.

Appendix

A. Fibonacci

```
-FunctionDecl 0x55ee47f78700 <x.c:3:1, line:15:1> line:3:5 main 'int ()'
 `-CompoundStmt 0x55ee47f7a060 <line:4:1, line:15:1>
   |-DeclStmt 0x55ee47f78980 <line:5:2, col:22>
   | |-VarDecl 0x55ee47f787b0 <col:2, col:9> col:6 used p1 'int' cinit
   | | `-IntegerLiteral 0x55ee47f78810 <col:9> 'int' 0
   | |-VarDecl 0x55ee47f78848 <col:2, col:14> col:11 used p2 'int' cinit
   | | `-IntegerLiteral 0x55ee47f788a8 <col:14> 'int' 0
   | `-VarDecl 0x55ee47f788e0 <col:2, col:21> col:16 used curr 'int' cinit
   |   `-IntegerLiteral 0x55ee47f78940 <col:21> 'int' 1
   |-ForStmt 0x55ee47f79ed8 <line:6:2, line:12:2>
   | |-DeclStmt 0x55ee47f78a30 <line:6:6, col:13>
   | | `-VarDecl 0x55ee47f789b0 <col:6, col:12> col:10 used i 'int' cinit
   | |   `-IntegerLiteral 0x55ee47f78a10 <col:12> 'int' 0
   | |-<<<NULL>>>
   | |-BinaryOperator 0x55ee47f78aa8 <col:14, col:16> 'int' '<'
   | | |-ImplicitCastExpr 0x55ee47f78a90 <col:14> 'int' <LValueToRValue>
   | | | `-DeclRefExpr 0x55ee47f78a48 <col:14> 'int' lvalue Var 0x55ee47f789b0 'i' 'int'
   | | `-IntegerLiteral 0x55ee47f78a70 <col:16> 'int' 10
   | |-UnaryOperator 0x55ee47f78af8 <col:19, col:20> 'int' postfix '++'
   | | `-DeclRefExpr 0x55ee47f78ad0 <col:19> 'int' lvalue Var 0x55ee47f789b0 'i' 'int'
   | `-CompoundStmt 0x55ee47f79ea8 <line:7:2, line:12:2>
   |   |-CallExpr 0x55ee47f79c10 <line:8:6, col:23> 'int'
   |   | |-ImplicitCastExpr 0x55ee47f79bf8 <col:6> 'int (*)(const char *, ...)' <FunctionToPointerDecay>
   |   | | `-DeclRefExpr 0x55ee47f78b18 <col:6> 'int (const char *, ...)' Function 0x55ee47f698d8 'printf' 'int (const char *, ...)'
   |   | |-ImplicitCastExpr 0x55ee47f79c60 <col:13> 'const char *' <BitCast>
   |   | | `-ImplicitCastExpr 0x55ee47f79c48 <col:13> 'char *' <ArrayToPointerDecay>
   |   | |   `-StringLiteral 0x55ee47f79b70 <col:13> 'char [4]' lvalue "%d "
   |   | `-ImplicitCastExpr 0x55ee47f79c78 <col:19> 'int' <LValueToRValue>
   |   |   `-DeclRefExpr 0x55ee47f79ba0 <col:19> 'int' lvalue Var 0x55ee47f788e0 'curr' 'int'
   |   |-BinaryOperator 0x55ee47f79cf8 <line:9:6, col:9> 'int' '='
   |   | |-DeclRefExpr 0x55ee47f79c90 <col:6> 'int' lvalue Var 0x55ee47f787b0 'p1' 'int'
   |   | `-ImplicitCastExpr 0x55ee47f79ce0 <col:9> 'int' <LValueToRValue>
   |   |   `-DeclRefExpr 0x55ee47f79cb8 <col:9> 'int' lvalue Var 0x55ee47f78848 'p2' 'int'
   |   |-BinaryOperator 0x55ee47f79d88 <line:10:6, col:9> 'int' '='
   |   | |-DeclRefExpr 0x55ee47f79d20 <col:6> 'int' lvalue Var 0x55ee47f78848 'p2' 'int'
   |   | `-ImplicitCastExpr 0x55ee47f79d70 <col:9> 'int' <LValueToRValue>
   |   |   `-DeclRefExpr 0x55ee47f79d48 <col:9> 'int' lvalue Var 0x55ee47f788e0 'curr' 'int'
   |   `-BinaryOperator 0x55ee47f79e80 <line:11:6, col:14> 'int' '='
   |     |-DeclRefExpr 0x55ee47f79db0 <col:6> 'int' lvalue Var 0x55ee47f788e0 'curr' 'int'
   |     `-BinaryOperator 0x55ee47f79e58 <col:11, col:14> 'int' '+'
   |       |-ImplicitCastExpr 0x55ee47f79e28 <col:11> 'int' <LValueToRValue>
   |       | `-DeclRefExpr 0x55ee47f79dd8 <col:11> 'int' lvalue Var 0x55ee47f787b0 'p1' 'int'
   |       `-ImplicitCastExpr 0x55ee47f79e40 <col:14> 'int' <LValueToRValue>
   |         `-DeclRefExpr 0x55ee47f79e00 <col:14> 'int' lvalue Var 0x55ee47f78848 'p2' 'int'
   |-CallExpr 0x55ee47f79fc8 <line:13:2, col:24> 'int'
   | |-ImplicitCastExpr 0x55ee47f79fb0 <col:2> 'int (*)(const char *, ...)' <FunctionToPointerDecay>
   | | `-DeclRefExpr 0x55ee47f79f10 <col:2> 'int (const char *, ...)' Function 0x55ee47f698d8 'printf' 'int (const char *, ...)'
   | `-ImplicitCastExpr 0x55ee47f7a010 <col:9> 'const char *' <BitCast>
   |   `-ImplicitCastExpr 0x55ee47f79ff8 <col:9> 'char *' <ArrayToPointerDecay>
   |     `-StringLiteral 0x55ee47f79f78 <col:9> 'char [11]' lvalue "\n\tThe End\n"
   `-ReturnStmt 0x55ee47f7a048 <line:14:2, col:9>
     `-IntegerLiteral 0x55ee47f7a028 <col:9> 'int' 0
```

B. Program with Function calls

```
FunctionDecl 0x56238d25f848 <x.c:3:1, line:8:1> line:3:5 used diff 'int (int, int)'
|-ParmVarDecl 0x56238d25f6f8 <col:10, col:14> col:14 used a 'int'
|-ParmVarDecl 0x56238d25f770 <col:16, col:20> col:20 used b 'int'
`-CompoundStmt 0x56238d260bb8 <line:4:1, line:8:1>
  |-IfStmt 0x56238d25fa78 <line:5:5, line:6:20>
  | |-<<<NULL>>>
  | |-<<<NULL>>>
  | |-BinaryOperator 0x56238d25f970 <line:5:8, col:11> 'int' '<='
  | | |-ImplicitCastExpr 0x56238d25f940 <col:8> 'int' <LValueToRValue>
  | | | `-DeclRefExpr 0x56238d25f8f0 <col:8> 'int' lvalue ParmVar 0x56238d25f6f8 'a' 'int'
  | | `-ImplicitCastExpr 0x56238d25f958 <col:11> 'int' <LValueToRValue>
  | |   `-DeclRefExpr 0x56238d25f918 <col:11> 'int' lvalue ParmVar 0x56238d25f770 'b' 'int'
  | |-ReturnStmt 0x56238d25fa60 <line:6:9, col:20>
  | | `-ParenExpr 0x56238d25fa40 <col:16, col:20> 'int'
  | |   `-BinaryOperator 0x56238d25fa18 <col:17, col:19> 'int' '-'
  | |     |-ImplicitCastExpr 0x56238d25f9e8 <col:17> 'int' <LValueToRValue>
  | |     | `-DeclRefExpr 0x56238d25f998 <col:17> 'int' lvalue ParmVar 0x56238d25f770 'b' 'int'
  | |     `-ImplicitCastExpr 0x56238d25fa00 <col:19> 'int' <LValueToRValue>
  | |       `-DeclRefExpr 0x56238d25f9c0 <col:19> 'int' lvalue ParmVar 0x56238d25f6f8 'a' 'int'
  | `-<<<NULL>>>
  `-ReturnStmt 0x56238d260ba0 <line:7:5, col:16>
    `-ParenExpr 0x56238d25fb58 <col:12, col:16> 'int'
      `-BinaryOperator 0x56238d25fb30 <col:13, col:15> 'int' '-'
        |-ImplicitCastExpr 0x56238d25fb00 <col:13> 'int' <LValueToRValue>
        | `-DeclRefExpr 0x56238d25fab0 <col:13> 'int' lvalue ParmVar 0x56238d25f6f8 'a' 'int'
        `-ImplicitCastExpr 0x56238d25fb18 <col:15> 'int' <LValueToRValue>
          `-DeclRefExpr 0x56238d25fad8 <col:15> 'int' lvalue ParmVar 0x56238d25f770 'b' 'int'
```

```
FunctionDecl 0x56238d260c30 <line:10:1, line:18:1> line:10:5 main 'int ()'
`-CompoundStmt 0x56238d2612f8 <line:11:1, line:18:1>
  |-DeclStmt 0x56238d260f70 <line:12:2, col:33>
  | |-VarDecl 0x56238d260ce0 <col:2, col:18> col:6 used d1 'int' cinit
  | | `-CallExpr 0x56238d260df0 <col:9, col:18> 'int'
  | |   |-ImplicitCastExpr 0x56238d260dd8 <col:9> 'int (*)(int, int)' <FunctionToPointerDecay>
  | |   | `-DeclRefExpr 0x56238d260d40 <col:9> 'int (int, int)' Function 0x56238d25f848 'diff' 'int (int, int)'
  | |   |-IntegerLiteral 0x56238d260d68 <col:14> 'int' 8
  | |   `-IntegerLiteral 0x56238d260d88 <col:16> 'int' 10
  | `-VarDecl 0x56238d260e40 <col:2, col:32> col:20 used d2 'int' cinit
  |   `-CallExpr 0x56238d260f20 <col:23, col:32> 'int'
  |     |-ImplicitCastExpr 0x56238d260f08 <col:23> 'int (*)(int, int)' <FunctionToPointerDecay>
  |     | `-DeclRefExpr 0x56238d260ea0 <col:23> 'int (int, int)' Function 0x56238d25f848 'diff' 'int (int, int)'
  |     |-IntegerLiteral 0x56238d260ec8 <col:28> 'int' 10
  |     `-IntegerLiteral 0x56238d260ee8 <col:31> 'int' 8
  |-IfStmt 0x56238d261288 <line:13:2, line:16:28>
  | |-<<<NULL>>>
  | |-<<<NULL>>>
  | |-BinaryOperator 0x56238d261008 <line:13:5, col:9> 'int' '=='
  | | |-ImplicitCastExpr 0x56238d260fd8 <col:5> 'int' <LValueToRValue>
  | | | `-DeclRefExpr 0x56238d260f88 <col:5> 'int' lvalue Var 0x56238d260ce0 'd1' 'int'
  | | `-ImplicitCastExpr 0x56238d260ff0 <col:9> 'int' <LValueToRValue>
  | |   `-DeclRefExpr 0x56238d260fb0 <col:9> 'int' lvalue Var 0x56238d260e40 'd2' 'int'
  | |-CallExpr 0x56238d261110 <line:14:6, col:25> 'int'
  | | |-ImplicitCastExpr 0x56238d2610f8 <col:6> 'int (*)(const char *, ...)' <FunctionToPointerDecay>
  | | | `-DeclRefExpr 0x56238d261030 <col:6> 'int (const char *, ...)' Function 0x56238d250908 'printf' 'int (const char *, ...)'
  | | `-ImplicitCastExpr 0x56238d261158 <col:13> 'const char *' <BitCast>
  | |   `-ImplicitCastExpr 0x56238d261140 <col:13> 'char *' <ArrayToPointerDecay>
  | |     `-StringLiteral 0x56238d261098 <col:13> 'char [10]' lvalue "It works\n"
  | `-CallExpr 0x56238d261228 <line:16:6, col:28> 'int'
  |   |-ImplicitCastExpr 0x56238d261210 <col:6> 'int (*)(const char *, ...)' <FunctionToPointerDecay>
  |   | `-DeclRefExpr 0x56238d261170 <col:6> 'int (const char *, ...)' Function 0x56238d250908 'printf' 'int (const char *, ...)'
  |   `-ImplicitCastExpr 0x56238d261270 <col:13> 'const char *' <BitCast>
  |     `-ImplicitCastExpr 0x56238d261258 <col:13> 'char *' <ArrayToPointerDecay>
  |       `-StringLiteral 0x56238d2611d8 <col:13> 'char [13]' lvalue "Not working\n"
  `-ReturnStmt 0x56238d2612e0 <line:17:2, col:9>
    `-IntegerLiteral 0x56238d2612c0 <col:9> 'int' 0
```

C. All .ll files can be found here. https://github.com/lancecorp72/Compilers-2/tree/master/Mini%20Asgn%201/LL%20files