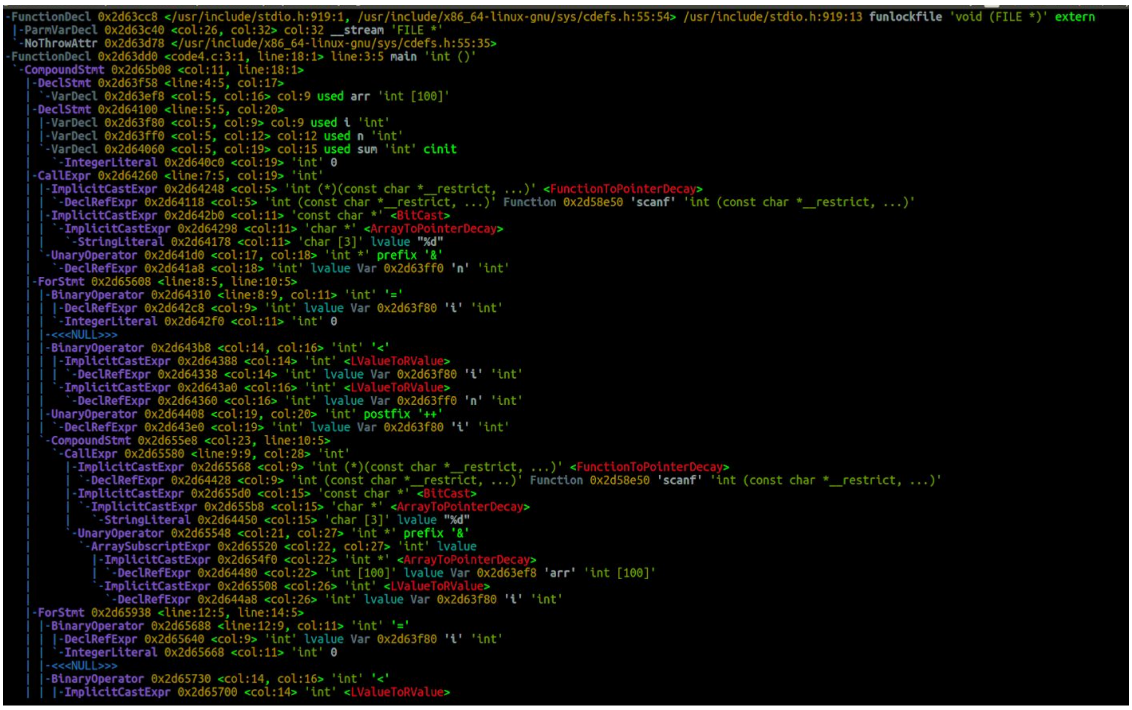
Compilers Mini Assignment 1

Vignesh K
ES17BTECH11023

Clang-AST Structure

Five programs were compiled in order to study the AST structure. Every AST had most of the code contained inside ***Translation Unit Declaration*** which had the code for functions and declarations in header files included. The program code was in the last 70-80 lines:

- **Sum of elements in an array program:** Tree starts 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 are the other nodes within the appropriate subtrees. Finally, the AST reaches ***Return Statement*** and terminates.



```
-FunctionDecl 0x2d63cc8 </usr/include/stdio.h:919:1, /usr/include/x86_64-linux-gnu/sys/cdefs.h:55:54> /usr/include/stdio.h:919:13 funlockfile 'void (FILE *)' extern
 |-ParmVarDecl 0x2d63c40 <col:26, col:32> col:32 __stream 'FILE *'
 `-NoThrowAttr 0x2d63d78 </usr/include/x86_64-linux-gnu/sys/cdefs.h:55:35>
-FunctionDecl 0x2d63dd0 <code4.c:3:1, line:18:1> line:3:5 main 'int ()'
 `-CompoundStmt 0x2d65b08 <col:11, line:18:1>
   |-DeclStmt 0x2d63f58 <line:4:5, col:17>
   | `-VarDecl 0x2d63ef8 <col:5, col:16> col:9 used arr 'int [100]'
   |-DeclStmt 0x2d64100 <line:5:5, col:20>
   | |-VarDecl 0x2d63f80 <col:5, col:9> col:9 used i 'int'
   | |-VarDecl 0x2d63ff0 <col:5, col:12> col:12 used n 'int'
   | `-VarDecl 0x2d64060 <col:5, col:19> col:15 used sum 'int' cinit
   |   `-IntegerLiteral 0x2d640c0 <col:19> 'int' 0
   |-CallExpr 0x2d64260 <line:7:5, col:19> 'int'
   | |-ImplicitCastExpr 0x2d64248 <col:5> 'int (*)(const char *__restrict, ...)' <FunctionToPointerDecay>
   | | `-DeclRefExpr 0x2d64118 <col:5> 'int (const char *__restrict, ...)' Function 0x2d58e50 'scanf' 'int (const char *__restrict, ...)'
   | |-ImplicitCastExpr 0x2d642b0 <col:11> 'const char *' <BitCast>
   | | `-ImplicitCastExpr 0x2d64298 <col:11> 'char *' <ArrayToPointerDecay>
   | |   `-StringLiteral 0x2d64178 <col:11> 'char [3]' lvalue "%d"
   | `-UnaryOperator 0x2d641d0 <col:17, col:18> 'int *' prefix '&'
   |   `-DeclRefExpr 0x2d641a8 <col:18> 'int' lvalue Var 0x2d63ff0 'n' 'int'
   |-ForStmt 0x2d65608 <line:8:5, line:10:5>
   | |-BinaryOperator 0x2d64310 <line:8:9, col:11> 'int' '='
   | | |-DeclRefExpr 0x2d642c8 <col:9> 'int' lvalue Var 0x2d63f80 'i' 'int'
   | | `-IntegerLiteral 0x2d642f0 <col:11> 'int' 0
   | |-<<<NULL>>>
   | |-BinaryOperator 0x2d643b8 <col:14, col:16> 'int' '<'
   | | |-ImplicitCastExpr 0x2d64388 <col:14> 'int' <LValueToRValue>
   | | | `-DeclRefExpr 0x2d64338 <col:14> 'int' lvalue Var 0x2d63f80 'i' 'int'
   | | `-ImplicitCastExpr 0x2d643a0 <col:16> 'int' <LValueToRValue>
   | |   `-DeclRefExpr 0x2d64360 <col:16> 'int' lvalue Var 0x2d63ff0 'n' 'int'
   | |-UnaryOperator 0x2d64408 <col:19, col:20> 'int' postfix '++'
   | | `-DeclRefExpr 0x2d643e0 <col:19> 'int' lvalue Var 0x2d63f80 'i' 'int'
   | `-CompoundStmt 0x2d655e8 <col:23, line:10:5>
   |   `-CallExpr 0x2d65580 <line:9:9, col:28> 'int'
   |     |-ImplicitCastExpr 0x2d65568 <col:9> 'int (*)(const char *__restrict, ...)' <FunctionToPointerDecay>
   |     | `-DeclRefExpr 0x2d64428 <col:9> 'int (const char *__restrict, ...)' Function 0x2d58e50 'scanf' 'int (const char *__restrict, ...)'
   |     |-ImplicitCastExpr 0x2d655d0 <col:15> 'const char *' <BitCast>
   |     | `-ImplicitCastExpr 0x2d655b8 <col:15> 'char *' <ArrayToPointerDecay>
   |     |   `-StringLiteral 0x2d64450 <col:15> 'char [3]' lvalue "%d"
   |     `-UnaryOperator 0x2d65548 <col:21, col:27> 'int *' prefix '&'
   |       `-ArraySubscriptExpr 0x2d65520 <col:22, col:27> 'int' lvalue
   |         |-ImplicitCastExpr 0x2d654f0 <col:22> 'int *' <ArrayToPointerDecay>
   |         | `-DeclRefExpr 0x2d64480 <col:22> 'int [100]' lvalue Var 0x2d63ef8 'arr' 'int [100]'
   |         `-ImplicitCastExpr 0x2d65508 <col:26> 'int' <LValueToRValue>
   |           `-DeclRefExpr 0x2d644a8 <col:26> 'int' lvalue Var 0x2d63f80 'i' 'int'
   |-ForStmt 0x2d65938 <line:12:5, line:14:5>
   | |-BinaryOperator 0x2d65688 <line:12:9, col:11> 'int' '='
   | | |-DeclRefExpr 0x2d65640 <col:9> 'int' lvalue Var 0x2d63f80 'i' 'int'
   | | `-IntegerLiteral 0x2d65668 <col:11> 'int' 0
   | |-<<<NULL>>>
   | |-BinaryOperator 0x2d65730 <col:14, col:16> 'int' '<'
   | | |-ImplicitCastExpr 0x2d65700 <col:14> 'int' <LValueToRValue>
```

- **Fibonacci Sequence: I**t is a recursive function. 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 fibonacci() has a similar tree structure like main(). The parameters are acknowledged at the start of the function as special *Variable.*

```
-FunctionDecl 0x29d3ec8 <code5.c:3:1, line:10:1> line:3:5 used Fibonacci 'int (int)'
 |-ParmVarDecl 0x29d3e40 <col:15, col:19> col:19 used n 'int'
 `-CompoundStmt 0x29d43d8 <col:21, line:10:1>
   `-IfStmt 0x29d43a8 <line:4:4, line:9:48>
     |-<<<NULL>>>
     |-BinaryOperator 0x29d3fd8 <line:4:9, col:14> 'int' '=='
     | |-ImplicitCastExpr 0x29d3fc0 <col:9> 'int' <LValueToRValue>
     | | `-DeclRefExpr 0x29d3f78 <col:9> 'int' lvalue ParmVar 0x29d3e40 'n' 'int'
     | `-IntegerLiteral 0x29d3fa0 <col:14> 'int' 0
     |-ReturnStmt 0x29d4020 <line:5:7, col:14>
     | `-IntegerLiteral 0x29d4000 <col:14> 'int' 0
     `-IfStmt 0x29d4378 <line:6:9, line:9:48>
       |-<<<NULL>>>
       |-BinaryOperator 0x29d4098 <line:6:14, col:19> 'int' '=='
       | |-ImplicitCastExpr 0x29d4080 <col:14> 'int' <LValueToRValue>
       | | `-DeclRefExpr 0x29d4038 <col:14> 'int' lvalue ParmVar 0x29d3e40 'n' 'int'
       | `-IntegerLiteral 0x29d4060 <col:19> 'int' 1
       |-ReturnStmt 0x29d40e0 <line:7:7, col:14>
       | `-IntegerLiteral 0x29d40c0 <col:14> 'int' 1
       `-ReturnStmt 0x29d4360 <line:9:7, col:48>
         `-ParenExpr 0x29d4340 <col:14, col:48> 'int'
           `-BinaryOperator 0x29d4318 <col:16, col:46> 'int' '+'
             |-CallExpr 0x29d41f0 <col:16, col:29> 'int'
             | |-ImplicitCastExpr 0x29d41d8 <col:16> 'int (*)(int)' <FunctionToPointerDecay>
             | | `-DeclRefExpr 0x29d40f8 <col:16> 'int (int)' Function 0x29d3ec8 'Fibonacci' 'int (int)'
             | `-BinaryOperator 0x29d4180 <col:26, col:28> 'int' '-'
             |   |-ImplicitCastExpr 0x29d4168 <col:26> 'int' <LValueToRValue>
             |   | `-DeclRefExpr 0x29d4120 <col:26> 'int' lvalue ParmVar 0x29d3e40 'n' 'int'
             |   `-IntegerLiteral 0x29d4148 <col:28> 'int' 1
             `-CallExpr 0x29d42e8 <col:33, col:46> 'int'
               |-ImplicitCastExpr 0x29d42d0 <col:33> 'int (*)(int)' <FunctionToPointerDecay>
               | `-DeclRefExpr 0x29d4220 <col:33> 'int (int)' Function 0x29d3ec8 'Fibonacci' 'int (int)'
               `-BinaryOperator 0x29d42a8 <col:43, col:45> 'int' '-'
                 |-ImplicitCastExpr 0x29d4290 <col:43> 'int' <LValueToRValue>
                 | `-DeclRefExpr 0x29d4248 <col:43> 'int' lvalue ParmVar 0x29d3e40 'n' 'int'
                 `-IntegerLiteral 0x29d4270 <col:45> 'int' 2
-FunctionDecl 0x29d4418 <line:12:1, line:26:1> line:12:5 main 'int ()'
 `-CompoundStmt 0x29d5ff8 <col:11, line:26:1>
   |-DeclStmt 0x29d5840 <line:13:4, col:19>
   | |-VarDecl 0x29d44d0 <col:4, col:8> col:8 used n 'int'
   | |-VarDecl 0x29d5730 <col:4, col:15> col:11 used i 'int' cinit
   | | `-IntegerLiteral 0x29d5790 <col:15> 'int' 0
   | `-VarDecl 0x29d57c0 <col:4, col:18> col:18 used c 'int'
   |-CallExpr 0x29d59a0 <line:15:4, col:17> 'int'
   | |-ImplicitCastExpr 0x29d5988 <col:4> 'int (*)(const char *__restrict, ...)' <FunctionToPointerDecay>
   | | `-DeclRefExpr 0x29d5858 <col:4> 'int (const char *__restrict, ...)' Function 0x29c8ed0 'scanf' 'int (const char *__restrict, ...)'
   | |-ImplicitCastExpr 0x29d59f0 <col:10> 'const char *' <BitCast>
   | | `-ImplicitCastExpr 0x29d59d8 <col:10> 'char *' <ArrayToPointerDecay>
   | |   `-StringLiteral 0x29d58b8 <col:10> 'char [3]' lvalue "%d"
   | `-UnaryOperator 0x29d5910 <col:15, col:16> 'int *' prefix '&'
   |   `-DeclRefExpr 0x29d58e8 <col:16> 'int' lvalue Var 0x29d44d0 'n' 'int'
   |-CallExpr 0x29d5ac0 <line:17:4, col:32> 'int'
```

- ➢ Trees were intended to separate different blocks/subtrees from each other. Objects of the same subtree are encapsulated together using a series of pipe symbols (‘|’).
- ➢ LLVM has three types of classes : declarations, statements and types.

Clang AST Traversal

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.

- ➢ It is a depth first traversal and all nodes are visited mostly once.

➢ It goes through the class hierarchy from dynamic type to a top-tier class.
➢ It doesn't enter each note but calls another function to visit the node.

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 may not be clear or be cut from code, in the form of the llvm_unreachable() function. Note that both of these do not abort the program when flagged.

LLVM-IR

➢ IR is a low-level programming language similar to 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.
➢ Below is the analysis of .ll files of a few non-trivial C programs containing functions, strings and loops. 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 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.

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. 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 the input into tokens.
- 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.
- 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.

- ➢ 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.

Appendix

The .ll files can be found here:

https://github.com/VickyakaKV/Compilers-2/tree/master/Mini%20Asn%201/LL%20Files