CS3423: Compilers - II

Mini-Assignment #3

Soumi Chakraborty

ES19BTECH11017

1 Clang-AST

Command to generate the AST:

```
clang -Xclang -ast-dump -fsyntax-only filename.c
```

1.1 Non-trivial programs and their ASTs

This section dissects and analyses the AST of one non-trivial C program, and other such programs are included in the gzip file attached with the assignment.

Code (factorial_loop.c):

```c
#include<stdio.h>

int main()
{
 int i, fact = 1, n;
 printf("Enter a number: ");
 scanf("%d", &n);
 for(i = 1; i <= n; i++)
 {
   fact = fact * i;
 }
 printf("Factorial of %d is: %d", n, fact);
 return 0;
}
```

AST:

```
`-FunctionDecl 0x61811a0 <factorial_loop.c:3:1, line:14:1> line:3:5 main 'int ()'
  `-CompoundStmt 0x6181b18 <line:4:1, line:14:1>
    |-DeclStmt 0x61813e8 <line:5:3, col:21>
    | |-VarDecl 0x6181258 <col:3, col:7> col:7 used i 'int'
    | |-VarDecl 0x61812d0 <col:3, col:17> col:10 used fact 'int' cinit
    | | `-IntegerLiteral 0x6181330 <col:17> 'int' 1
    | `-VarDecl 0x6181368 <col:3, col:20> col:20 used n 'int'
    |-CallExpr 0x61814a0 <line:6:3, col:28> 'int'
    | |-ImplicitCastExpr 0x6181488 <col:3> 'int (*)(const char *, ...)' <FunctionToPointerDecay>
    | | `-DeclRefExpr 0x6181400 <col:3> 'int (const char *, ...)' Function 0x6171040 'printf' 'int (const char *, ...)'
    | `-ImplicitCastExpr 0x61814e8 <col:10> 'const char *' <BitCast>
    |   `-ImplicitCastExpr 0x61814d0 <col:10> 'char *' <ArrayToPointerDecay>
    |     `-StringLiteral 0x6181428 <col:10> 'char [17]' lvalue "Enter a number: "
    |-CallExpr 0x6181620 <line:7:3, col:17> 'int'
    | |-ImplicitCastExpr 0x6181608 <col:3> 'int (*)(const char *restrict, ...)' <FunctionToPointerDecay>
    | | `-DeclRefExpr 0x6181500 <col:3> 'int (const char *restrict, ...)' Function 0x6174d00 'scanf' 'int (const char *restrict, ...)'
    | |-ImplicitCastExpr 0x6181670 <col:9> 'const char *' <BitCast>
    | | `-ImplicitCastExpr 0x6181658 <col:9> 'char *' <ArrayToPointerDecay>
    | |   `-StringLiteral 0x6181568 <col:9> 'char [3]' lvalue "%d"
    | `-UnaryOperator 0x61815c0 <col:15, col:16> 'int *' prefix '&' cannot overflow
    |   `-DeclRefExpr 0x6181598 <col:16> 'int' lvalue Var 0x6181368 'n' 'int'
    |-ForStmt 0x61818f8 <line:8:3, line:11:3>
    | |-BinaryOperator 0x61816d0 <line:8:7, col:11> 'int' '='
    | | |-DeclRefExpr 0x6181688 <col:7> 'int' lvalue Var 0x6181258 'i' 'int'
    | | `-IntegerLiteral 0x61816b0 <col:11> 'int' 1
    | |-<<<NULL>>>
    | |-BinaryOperator 0x6181778 <col:14, col:19> 'int' '<='
    | | |-ImplicitCastExpr 0x6181748 <col:14> 'int' <LValueToRValue>
    | | | `-DeclRefExpr 0x61816f8 <col:14> 'int' lvalue Var 0x6181258 'i' 'int'
    | | `-ImplicitCastExpr 0x6181760 <col:19> 'int' <LValueToRValue>
    | |   `-DeclRefExpr 0x6181720 <col:19> 'int' lvalue Var 0x6181368 'n' 'int'
    | |-UnaryOperator 0x61817c8 <col:22, col:23> 'int' postfix '++'
    | | `-DeclRefExpr 0x61817a0 <col:22> 'int' lvalue Var 0x6181258 'i' 'int'
    | `-CompoundStmt 0x61818e0 <line:9:3, line:11:3>
    |   `-BinaryOperator 0x61818b8 <line:10:5, col:19> 'int' '='
    |     |-DeclRefExpr 0x61817e8 <col:5> 'int' lvalue Var 0x61812d0 'fact' 'int'
    |     `-BinaryOperator 0x6181890 <col:12, col:19> 'int' '*'
    |       |-ImplicitCastExpr 0x6181860 <col:12> 'int' <LValueToRValue>
    |       | `-DeclRefExpr 0x6181810 <col:12> 'int' lvalue Var 0x61812d0 'fact' 'int'
    |       `-ImplicitCastExpr 0x6181878 <col:19> 'int' <LValueToRValue>
    |         `-DeclRefExpr 0x6181838 <col:19> 'int' lvalue Var 0x6181258 'i' 'int'
    |-CallExpr 0x6181a40 <line:12:3, col:43> 'int'
    | |-ImplicitCastExpr 0x6181a28 <col:3> 'int (*)(const char *, ...)' <FunctionToPointerDecay>
    | | `-DeclRefExpr 0x6181930 <col:3> 'int (const char *, ...)' Function 0x6171040 'printf' 'int (const char *, ...)'
    | |-ImplicitCastExpr 0x6181a98 <col:10> 'const char *' <BitCast>
    | | `-ImplicitCastExpr 0x6181a80 <col:10> 'char *' <ArrayToPointerDecay>
    | |   `-StringLiteral 0x6181998 <col:10> 'char [23]' lvalue "Factorial of %d is: %d"
    | |-ImplicitCastExpr 0x6181ab0 <col:36> 'int' <LValueToRValue>
    | | `-DeclRefExpr 0x61819d8 <col:36> 'int' lvalue Var 0x6181368 'n' 'int'
    | `-ImplicitCastExpr 0x6181ac8 <col:39> 'int' <LValueToRValue>
    |   `-DeclRefExpr 0x6181a00 <col:39> 'int' lvalue Var 0x61812d0 'fact' 'int'
    `-ReturnStmt 0x6181b00 <line:13:3, col:10>
      `-IntegerLiteral 0x6181ae0 <col:10> 'int' 0
```

From the AST, we can clearly see the different tokens which make up the program:

- ***FunctionDecl***: the main() function
- ***CompoundStmt***: the block which makes up the body of the main() function, and then later the for loop
- ***BinaryOperator***: the “<=” sign in the for loop
- ***ForStrt***: start of the for loop (this is different in other programs like the one with the switch case (***SwitchStrt***) and the enum)

Also note that the indentation levels are indicative of the parent and child components of the tree. Each part of each statement has been referred to in the AST and the indentation levels make it very easy to read and understand.

2 LLVM-IR

Command to generate the AST:

```
clang -S -emit-llvm filename.c
```

The command above generates a .ll file which contains the LLVM-IR code and they can all be found in the attached folder.

2.1 What is LLVM-IR?

LLVM-IR is the intermediate representation of the source code. Generating an IR makes the code platform independent, and a lot of processes like code optimisation can be done on the IR the same way regardless of the platform.

2.2 Observations

On going through the .ll files, the following observations were made:

- The i32's show that the variable types are decided at compile time and once assigned they usually remain that way until there is need for it to be type casted
- The variables, etc. in the code are assigned registers in the .ll file
- The variables are differentiated on the basis of their scope
- The flow of control is mentioned in this file as well

3 Assembly Language

Command to generate the assembly language codes:

```
clang -S -masm=intel filename.c
```

The command above generates a .s file which contains the assembly code and they can all be found in the attached folder.

3.1 What is assembly language?

Assembly language is a low level language that isn't meant to be easily readable by humans. Its main objective is to be able to run on the computer's hardware. It is the language that connects the high-level language like C to the hardware where the operations finally take place.

3.2 Observations

On going through the .s files, the following observations were made:

- We can observe keywords like 'xor' which tell the hardware what logical/mathematical operation to execute
- The keyword 'mov' is used for assignments
- All the statements match the expected syntax for assembly language:
    - label mnemonic operands comments

All the .s files can be found in the gzip file submitted with the assignment.