CS3423: Compilers - II

Mini Assignment #1

Saksham Mittal

Contents

1 AST Structure

1.1 Overview of AST

The AST is divided in three core classes:
1. **Declarations**
2. **Statements**
3. **Types**

They do not inherit from a common base class. So, there is no common interface for visiting all the nodes in the tree. Each node has dedicated traversal methods that allow you to navigate the tree.

AST can be generated using the following command:

```
./clang++ -Xclang -ast-dump -fsyntax-only test.cpp
```

1.2 Common expressions in the AST structure

Figure 1: Sample AST

These major statements occurred in the AST's.

1.2.1 if-stmt

It consists of a conditional Expr(BinaryOperator) and two Stmts, one for the then-case and one for the else-case respectively.

1.2.2 CompoundStmt

Compound statements are made up of two or more program statements that are executed together.

1.2.3 DeclStmt

This is used for declaration and initialization of variables.
It consists of:

1. Declaration part:

```
VarDecl 0x55f7089c0140 <col:5, col:9> col:9 used n 'int'
```

2. Initialization part:

```
IntegerLiteral 0x55f7089c0218  <col:19> 'int' 0
```

1.2.4 FucntionDecl

This denotes the function along with its parameters and return type.

1.2.5 for-stmt

This consists of declaration step, condition step, and increment step.
If consists of:

1. This declares an *int i*.

```
DeclStmt 0x55f7089c04b0 <line:7:9, col:16>
```

2. This acts as an condtional statement.

```
BinaryOperator 0x55f7089c1580  <col:18, col:20>  'int' '<'
```

3. This is the increment step.

```
UnaryOperator 0x55f7089c15d0 <col:23, col:24>'int' postfix '++'
```

1.2.6 printf-stmt

Using ImplicitCastExpr() for casting string to a constant char* from a char array as a pointer with DeclRefExpr() referencing to **printf**.

```
ImplicitCastExpr 0x55f7089c17b0 <col:13>'int (*)(const char *, ...)'
<FunctionToPointerDecay>
```

2 AST Traversals

Visitor Pattern : It is a technique (Object Oriented pattern) to separate data structure definitions from code that traverses the structure of AST.

2.1 Create a FrontendAction

FrontendAction is an interface that allows execution of user specific actions as part of the compilation.

2.2 Create a ASTConsumer

ASTConsumer is an interface used to write generic actions on an AST, regardless of how the AST was produced.

2.3 Use the RecursiveASTVisitor

The Recursive AST Visitor enables you to traverse the nodes of Clang AST in a depth-first manner. We visit specific nodes by extending the class and implementing the desired *Visit** method

2.4 Accessing the SourceManager and ASTContext

Information like source locations and global identifier information, are not stored in the AST nodes themselves, but in the ASTContext and its associated source manager. To retrieve them we need to hand the ASTContext into our RecursiveASTVisitor implementation.

3 Error messages

The **assert** mechanism is used to check the preconditions and assumptions, which might be caught early, hence reducing the debugging time significantly.

If the first argument of the assert turns out to be false, it will terminate the program(preferably giving a message).
Example:

```
inline Value *getOperand(unsigned I) {
  assert(I < Operands.size() && "getOperand() out of range!");
  return Operands[I];
}
```

In this example, if the argument size is greater than **Operands.size()**, the program is terminated and an error message is printed "*getOperand() out of range!*".

Assertions can also be made to indicate a peice of code that should not be

reached. With **assertions enabled**, the following will print the message if ever reached, and exits:

```
llvm_unreachable("Code should not be reached here.");
```

But we should remember that it both assert and llvm_unreachable wont exit the program in the release mode.

4 LLVM-IR

LLVM front end converts the OpenCL program into **LLVM IR**, an intermediate presentation that is intended to be used as input for the LLVM compiler back end. The back end is then responsible for translating the LLVM IR into target specific language which is usable by the parallel computing capable hardware, for example GPU.

To get the LLVM-IR, the following command is run:

```
./clang -S -emit-llvm test.cpp -o test.ll
```

This is what an *.ll* file looks like:

```
; ModuleID = 'test.cpp'
source_filename = "test.cpp"
target datalayout = "e-m:e-i64:64-f80:128-n8:16:32:64-S128"
target triple = "x86_64-unknown-linux-gnu"

; Function Attrs: noinline nounwind optnone uwtable
define dso_local void @_Z3sumii(i32 %a, i32 %b) #0 {
entry:
  %a.addr = alloca i32, align 4
  %b.addr = alloca i32, align 4
  %c = alloca i32, align 4
  store i32 %a, i32* %a.addr, align 4
  store i32 %b, i32* %b.addr, align 4
  %0 = load i32, i32* %a.addr, align 4
  %1 = load i32, i32* %b.addr, align 4
  %add = add nsw i32 %0, %1
  store i32 %add, i32* %c, align 4
  ret void
}

; Function Attrs: noinline norecurse nounwind optnone uwtable
define dso_local i32 @main() #1 {
entry:
  %retval = alloca i32, align 4
  store i32 0, i32* %retval, align 4
```

```
  call void @_Z3sumii(i32 1, i32 2)
  ret i32 0
}
```

Some main highlights of LLVM IR are:

4.1 Comments

Comments in LLVM assembly begin with a semicolon () and continue to the end of the line.

4.2 Declaration of Function

To declare a function, begin the declaration with the declare keyword followed by the return type, the function name, and an optional list of arguments to the function.
The declaration must be in the global scope. Eg.

```
declare i32 puts(i8*)
```

4.3 Defining a Function

To define a function, begin with the define keyword followed by the return type, and then the function name. Eg.

```
define i32 @main()
```

4.4 Calling a Function

To call the function, call **<function return type><function name><optional function arguments>**

4.5 Returning a Function

Each function ends with a return statement. There are two forms of return statement: **ret <type><value>** or **ret void**.

4.6 Declaring a vector/array

You declare a vector or array type as [**no. of elements X size of each element**].

4.7 Global identifiers

Global identifiers begin with the at (@) character. All function names and global variables must begin with **@**, as well.

4.8 Local identifiers

Local identifiers in the LLVM begin with a percent symbol (%). The typical regular expression for identifiers is **[%@][a.zA.Z$._][a.zA.Z$._0.9]***.

4.9 Declaring global string

You declare a global string constant for the myname string as follows: **@myname = constant [13x i8] c"Hello World!\00"**
Example from code:

```
@.str = private unnamed_addr constant [3 x i8] c"%d", align 1
```

4.10 For conditionals

It is divided into 3 parts:
for.cond: - The conditional statement of *for*.
for.body: - The body of the *for* statement.
for.inc: - The increment condition of *for* statement.

5 Assembly language

Consider a simple cpp program:

```
void sum(int a, int b, int c) {
    int d = a + b + c;
}
void sum(int a, int b) {
    int c = a + b;
}
int main() {
    sum(1, 2, 3);
    sum(1, 2);
    return 0;
}
```

To get the assembly code, the following command is run:

```
./clang -S test.cpp -o test.s
```

The assembly code generated for a cpp program with function overloading is:

```
        .file       "test.cpp"
        .text
        .globl       _Z3sumiii
        .type       _Z3sumiii, @function
_Z3sumiii:
```

```
.LFB0:
        .cfi_startproc
        pushq        %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq        %rsp, %rbp
        .cfi_def_cfa_register 6
        movl        %edi, -20(%rbp)
        movl        %esi, -24(%rbp)
        movl        %edx, -28(%rbp)
        movl        -20(%rbp), %edx
        movl        -24(%rbp), %eax
        addl        %eax, %edx
        movl        -28(%rbp), %eax
        addl        %edx, %eax
        movl        %eax, -4(%rbp)
        nop
        popq        %rbp
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
.LFE0:
        .size        _Z3sumiii, .-_Z3sumiii
        .globl        _Z3sumii
        .type        _Z3sumii, @function
_Z3sumii:
.LFB1:
        .cfi_startproc
        pushq        %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq        %rsp, %rbp
        .cfi_def_cfa_register 6
        movl        %edi, -20(%rbp)
        movl        %esi, -24(%rbp)
        movl        -20(%rbp), %edx
        movl        -24(%rbp), %eax
        addl        %edx, %eax
        movl        %eax, -4(%rbp)
        nop
        popq        %rbp
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
.LFE1:
        .size        _Z3sumii, .-_Z3sumii
```

```
        .globl        main
        .type        main, @function
main:
.LFB2:
        .cfi_startproc
        pushq        %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq        %rsp, %rbp
        .cfi_def_cfa_register 6
        movl        $3, %edx
        movl        $2, %esi
        movl        $1, %edi
        call        _Z3sumiii
        movl        $2, %esi
        movl        $1, %edi
        call        _Z3sumii
        movl        $0, %eax
        popq        %rbp
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
.LFE2:
        .size        main, .-main
        .ident        "GCC: (Ubuntu 7.3.0-16ubuntu3) 7.3.0"
        .section        .note.GNU-stack,"",@progbits
```

C++ supports function overloading, i.e., there can be more than one functions with same name and differences in parameters.
This technique of adding additional information to function names is called **Name Mangling**. C++ standard doesn't specify any particular technique for name mangling, so different compilers may append different information to function names.

The two functions sum are referenced by same name, but differs in the number of parameters(One has 2 and another has 3 parameters).

clang adds a **Z** followed by length of function name in int followed by the function name and then followed by argument type's abbreviation.

In the example the compiler differentiates them by referring them as:
_Z3sumiii - Here the 3 i's show that the number of parameters are 3.
_Z3sumii - Here there are 2 i's, which show that number of parameters are 2.

6 Compiler toolchain and options

6.1 Compiler Frontend - clang

-emit-ast : Generate the clang AST files.
-emit-llvm : Use llvm representation for assembler and object files.
-c : Only run preprocess, compile, and assemble steps.
-E : Only run the preprocessor.

6.2 Compiler Middle end

llvm-as : Converts a LLVM assembly file into LLVM bitcode.
llvm-dis : Converts LLVM bitcode to LLVM assembly language.

6.3 Optimizer

-dot-callgraph : It prints the call graph.
-dot-cfg : This prints the cfg graph.
-O1, -O2, -O3 : These options are for optimization levels, in increasing order of aggressiveness.
-Osize : This optimizes the binary for file size.
–**ffast-math** : This enables lossy optimization for floating point operations.

7 Kaleidoscope

Kaleidoscope is a procedural language that allows you to use conditionals, define functions, math, etc.
The only data type in Kaleidoscope is a 64-bit floating point type. Hence, the language doesn't require type declarations.
Example code:

```
# Compute the x'th fibonacci number.
def fib(x)
  if x < 3 then
    1
  else
    fib(x-1)+fib(x-2)

# This expression will compute the 40th number.
fib(40)
```

The external library functions are used by using the '**extern**' keyword before the funciton name.

7.1 Lexer for Kaleidoscope

The **Lexer** breaks the input into "*tokens*". Each token contains some metadata and a token code.
This is what it looks like:

```
// The lexer returns tokens [0-255] if it is an unknown character, otherwise one
// of these for known things.
enum Token {
  tok_eof = -1,

  // commands
  tok_def = -2,
  tok_extern = -3,

  // primary
  tok_identifier = -4,
  tok_number = -5,
};

static std::string IdentifierStr; // Filled in if tok_identifier
static double NumVal;             // Filled in if tok_number
```

The implementation of lexer function is done using a single function *gettok()*.
The definition is as follows:

```
// gettok - Return the next token from standard input.
static int gettok() {
  static int LastChar = ' ';

  // Skip any whitespace.
  while (isspace(LastChar))
    LastChar = getchar();
```

The **gettok** function is called to return the next token from standard input.

References

http://llvm.org/docs/GettingStarted.htmlcheck-here
http://clang.llvm.org/docs/IntroductionToTheClangAST.html
https://llvm.org/docs/CodingStandards.htmlassert-liberally
http://llvm.org/docs/tutorial/LangImpl01.html
https://en.wikipedia.org/wiki/Name_mangling
https://gist.github.com/zchee/48be861fcc3c5239ec1954b34d9a5205
https://gist.github.com/zchee/48be861fcc3c5239ec1954b34d9a5205