CS 502 – Fall 2012

Project 1 Unparsing

Deadline: Monday, October 22, 2012, 11:59pm

1. Project Description

1.1 Overview

This project requires to implement an **unparser** which traverses the abstract syntax tree (AST) of C language in GCC-4.7.0, and, based on which, regenerates the C source code.

AST is a high level intermediate representation. This project will enhance students' understanding of the implementation of a programming language, by knowing how a program can be represented in a compiler.

An unparser is useful for various purposes. For example, it can be used to beautify a program by reprinting a program in a uniform style, hopefully easier to read. It can be used to automatically insert statements for program testing and performance profiling, to perform source-level transformations for performance and portability reasons.

In order to reduce the mundane aspects and the overall load of the project, only a subset of C is required (see Appendix). However, students are welcome to implement a bigger set of C in the unparser.

There are two milestones in Project1.

Milestone 1: Deadline **October 10, 2012, 11:59 pm**(2.5weeks). Implement an unparser for the subset of C described in Appendix A.

Milestone 2: Deadline **October 22, 2012, 11:59 pm**(1.5 weeks). Extend the unparser such that it covers both the subset of C language in Appendix A and that in B.

The project is to be done individually, using any of the XINU machines in the CS department.

1.2 GCC Overview

GCC stands for “GNU Compiler Collection”. It is an integrated distribution of open-source compilers for several major programming languages, including C, C++, Java, and so on. For each language, there is a compiler, which is responsible for translating the source file into

possibly optimized target machine code. For example, C compiler is "cc1" and Java compiler is "jc1". CS502 projects will focus on C compiler (i.e. cc1).

Illustrated in Fig 1, the architecture of GCC is divided into the following three main parts:

- **Front-End**: source language-dependent.
- **Middle-End**: both language-independent and target-independent.
- **Back-End**: target-dependent.

The project will be done at the end of the Front-End stage.

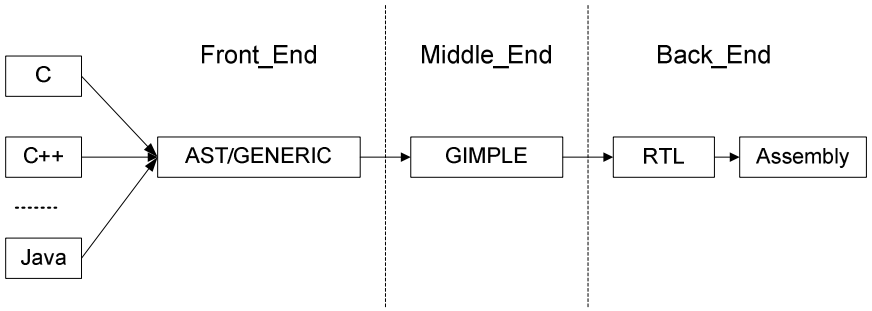

Fig 1. GCC internal framework

1.3 Project Call Chain in GCC-4.7.0

Fig 2 shows the main function call chain in GCC-4.7.0 source code related to Project1. The function call "cs502_proj1()", which should be implemented in Project1, starts the unparsing procedure.

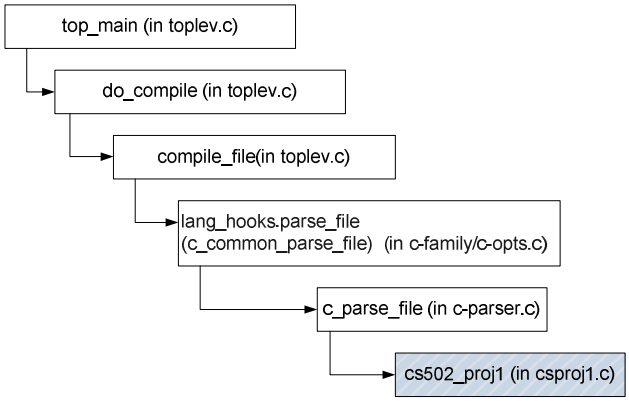

Fig 2. Where project1 is invoked

1.4 Tree in GCC4.7.0

It is crucial for students to get familiar with the data structure of the AST (known as GENERIC in GCC's terminology) in gcc-4.7.0. The central data structure is **tree (**defined in **tree.h, tree.def)**. A **tree** is a pointer type, but the object to which it points may be of a variety of types. For each tree node, it has a TREE_CODE to indicate what kind of tree node it is. For example, a function is represented by a tree fn with

TREE_CODE(fn) = FUNCTION_DECL

And the high-level organization of fn (the function) approximately looks like what Fig 3 shows, where each node represents a tree node. More details of tree nodes please refer to the resources listed in Part 5 of this document.

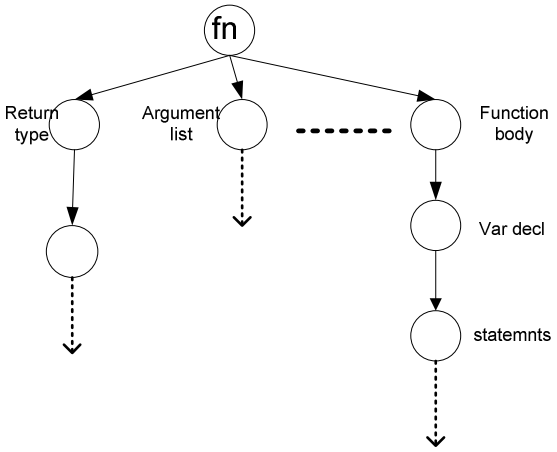

Fig 3. A possible high level organization of a tree with FUNCTION_DECL code

1.5 Other Suggestions

1) How to traverse all functions?

After the c parser, a call graph is build, represented by a global variable:

struct cgraph_node *cgraph_nodes

Each node in the call graph represent a function (Please refer to source code: cgraph.h). The following sample code snippet shows how to traverse the call graph and get every function:

```
struct cgraph_node *node;
 tree fn;
 for (node = cgraph_nodes; node; node = node->next) {
    fn = node->decl;//get a function
    ****
 }
```

2) How to handle global variable declaration?

In GCC-4.7.0, all global variables are stored in a global variable

struct varpool_node *varpool_nodes

The definition can be found in cgraph.h, varpool.c. The following code snippet shows how to traverse all global declarations:

```
Struct varpool_node *node;
For (node = varpool_nodes; node; node = node->next) {
        Tree decl = noe->decl; //get the global declaration
}
```

3) How to traverse and output a tree node?

This is exactly what students must investigate by doing this project However, to get a hint, one can take a look at the function **dump_generic_node** in **tree-pretty-print.c** file, which will give you a great example. DO **NOT** copy any code from those files! You must understand how to traverse the tree and write your own code. Understanding the TREE structure is not only crucial for Project1, but also for Project 2.

2. How to Start

To start the project, students need to follow the steps:

1. To set up the environment of Project1, copy and unzip the file “cs502_fall12.tar.gz" from the directory “/u/data/u3/cs502/Fall12/CS502”. A directory “cs502_fall12" will be created under students’ local directory, in which a soft link “gcc-4.7.0_src” and a directory “install” are included. The soft link “gcc-4.7.0_src” leads to gcc-4.7.0 source code directory on xinu, which can be read only. The “install” directory is students’ working directory.
2. Go to the directory “cs502_fall12/install/". (The entire set of GCC 4.7.0 files is very big. Therefore we share as many files as possible in order to reduce the storage requirement.) In this directory, students should write a source file named “csproj1.c" to implement the unparser. Now a template with an empty function cs502_proj1 and a sample Makefile is provided (how cs502_proj1 is invoked is described in section 1.3). After the unparser is written, run the command “make" to create your own version of program “cc1" which has the unparsing feature built in.

   Note: The project should **NOT** modify any existing source files in GCC4.7.0, as provided in the environment described above. Instead, a new source “csproj1.c" (and other new

source files you write) should be created. Again, the “csproj1.c" file should contain a function “void cs502_ proj1()" from which starts the unparsing. Students should add the new source files to the Makefile provided by the TA in the tar file mentioned above, but please do not change the other parts.

3. After a successful make, please run “.\cc1 some-file-name.c" to parse and then unparsed some-file-name.c. The program cc1 reads the source code, construct the AST, symbol table and call graph, and then, using your implemented unparsing feature, to regenerate the source code.
4. The two versions of source code may look different, but are equivalent. Five sample test files are provided under “/cs502_fall12/install” directory. Test1.c and test2.c are for Milestone1, while the rest two are for Milestone2. When the projects are submitted, the TA will also change those files somewhat for grading.

3. How to submit

Use the following commands to submit your files,

turnin –c cs502 –p proj1_m1 [your working directory] (for milestone 1)

turnin –c cs502 –p proj1_m2 [your working directory] (for milestone 2)

Both submissions should include:

1) All new source files.
2) The new Makefile.
3) A README file explaining anything you want the TA to know in order to re-produce your directory and generate your own cc1.
4) A project report in plain text (please see 6.Grading Criteria – Documentation)

Please be sure to run “**make clean**” before submitting your working directory.

4. Grading Criteria

The following criteria are used to grade both Milestone1 and Milestone2. Milestone1 will take 60% credits of Project1, and Milestone2 will take 40%.

Correctness: (70%)

The C source codes generated by the unparser must be compiled correctly and produce the same results as the input C programs (The warning messages can be ignored). *All submitted programs will be automatically checked against each other and against unparsers written in past years to detect potential instances of plagiarism. (Plagiarism will receive stiff penalties ranging from receiving an F for the project to an F for the entire course, depending on the severity of offence. The definition of plagiarism for this project is clear: the students are not allowed to copy others' code.)*

Readability: (10%)

The generated program must be reasonably formatted (i.e. indentation and line-separating) to make it readable.

Documentation: (20%)

Proper comments must be inserted in the new source files if they help to read your code.

In addition, each student submit a project report which

a. describes which parts of the project (referring to the language subset) are completed and which parts are not, and

b. lists all the files you have created and explain those new functions and major new data structures (if any).

5. Resources for Project1

There are resources useful for Project1, especially the second one.

1) GCC website

http://gcc.gnu.org/

2) GCC 4.7 internal manual, Tree:

http://gcc.gnu.org/onlinedocs/gccint/index.html#toc_GENERIC

3) GCC 4.7 Source code: linked from “gcc-4.7.0_src”.

6. Appendix: The Subset of ANSI-C for Project 1

(This description is subject to minor changes to clarify any doubts raised by students)

Appendix A: Milestone 1

A.1 Block Structures

In this subset, a C program has a main function and zero or more other functions, all of which are placed in a **single** source file. The organization of a C program looks like this:

```
< global declarations of types and variables>
<type> function-1 (<parameter list>)
{
        <block body>
}
…..
<type> function-n (<parameter list>)
{
        <block body>
}
```

There are **NO** header files included (such as "#include<stdio.h>"), **No** global declarations between any two functions.

<block body> consists of local declarations and a list of **ASSIGNMENT** statements, **FUNCTION** calls, **RETURN** statements .

In other words, only sequential programming structures are required in Milestone1.

A.2 Expression

For expressions in Milestone 1, we have the following operators:

= || && ! == != < > <= >= + - * /

The atomic operands are **ID**s, **integers**, **real numbers** and **characters**.

A.3 Data types and declarations

Scalar types: **integer**, **float** and **character**.

Variables may be initialized in their declaration statements.

Appendix B: The subset of ANSI-C for Project 1, Milestone 2

In this part, more program constructs are to be unparsed.

B.1 Statements

The statements include: **IF** statements (may contains a ELSE branch), **SWITCH** statements, **WHILE** statements , and **BLOCK**s.

A BLOCK looks like:

```
{
        <block body>
}
```

Note: a WHILE statement will be represented by **IF** tree nodes and **GOTO** tree nodes in AST. Under this circumstance, your unparser will generate the equivalent code using "if" and "goto" statements, rather than the same "while" structure.

B.2 Expression

In Milestone 2, pointer-related operations are required, such as

*p, &a, c->d.

B.3 Data types and declarations

High level types: **Arrays** and **Structures**

The following declaration statements remain in the language subset, no matter whether they are global or local.

```
struct a {
...
} xx;
struct a yy;
```

Appendix C: What are not required

In general, what is not explicitly stated in Appendix A and B will not be required.

To clarify potential doubts, we explicitly state that there is NO:

1. Unions
2. Pointer arithmetic
3. Typedef statements
4. Type coercion, such as " f = (float) a;"
5. Included header files, such as "#include <stdio.h>"