Homework 4

Compilers, Monsoon 2020

Swaraj Renghe
20171119

08 September 2020

1. Completed reading assignment. This is the `swap` subroutine.

```
extern int buf[];
int *bufp0 = &buf[0];
int *bufp1;

void swap() {
    int temp;

     bufp1 = &buf[1];
     temp = *bufp0;
    *bufp0 = *bufp1;
    *bufp1 = temp;
}
```

Question 7.12

Line Number in Fig. 7.10	Address	Value
15	0x80483cb	0x804945c
16	0x80483d0	0x8049458
18	0x80483d8	0x8049548
18	0x80483dc	0x8049458
23	0x80483e7	0x8049548

Question 7.15

A) To find the object files of `libc.a` and `libm.a`, we first find the original files. This can be done by running `gcc --print-file-name=libc.a` and `–print-file-name=libm.a`. Now that we have found the files, we can find the number of object files in this by running `ar -t <file_path> | wc -l` .at Thus, on my local installation, I have **1690 object files in `libc.a`** and **484 object files in `libm.a`**

B) The -g flag in gcc enables debugging. It produces debugging information in the operating system's native format, which can then be used by GDB. On most systems that use stabs format, -g enables use of extra debugging information that only GDB can use; this extra information makes debugging work better in GDB but probably makes other debuggers crash or refuse to read the program. If we compile a simple addition program using the two options, the assembly output of the main function for both programs is identical, but the -g option is also accompanied by the original source code, for easy debugging. In addition, the disassembly of section `.note.gnu.build-id` is different too, and the -g version is accompanied with some extra debugging information at the end of the file.

C) On my local installation, when I run ldd on a compiled program, I can see the shared libraries being used in the program. In my case, these are

- linux-vdso - a small shared library that the kernel automatically maps into the address space of all user-space applications.
- libc.so – the standard C library for UNIX / LINUX systems.
- ld-linux-x86-64.so - the dynamic linker for the C language for UNIX/LINUX systems.

2. Completed reading assignment. We need to prove that if a grammar is parse tree-ambiguous then it is right-ambiguous. We can start by proving a simple lemma first.

   For every parse tree, there is a unique rightmost derivation.

   Proof We can use induction on the height (h) of the parse tree to prove this lemma.
   If the height is taken as 1, then it means that it has all terminals in the next step, therefore only a single derivation is possible.
   Let us assume the lemma holds for all trees of height h. Consider a parse tree of height h+1. Take the subtree (S) of height h i.e, the tree with the lowest height nodes removed.
   Then, ∃ a unique rightmost derivation for this subtree. S =>* X1 X2...Xn Clearly, the last node just consists of the terminals and so we can expand them starting from the right-hand side to get a unique derivation.

   Building on this idea, if we assume that a grammar G is parse-tree ambiguous i.e, ∃ a string k such that k has 2 parse trees. We want to prove that G is right ambiguous i.e, ∃ a string k such that k has 2 right derivations.

Since there are 2 parse trees for string k, there are also 2 unique rightmost derivations using Lemma. As a result, the grammar is right-ambiguous.

3. We can define the CFG for the grammar give like -

   ```
   G = <A, B, C, D> where
   A = {Z, I}
   B = {a, b, *, ·, +, (, )}
   C = Z
   ```

   D is the set of the production rules given by -

   - `Z → I | Z* | Z·Z | Z+Z | (Z);`
   - `I → a | b;`

   Unfortunately, the above grammar is ambiguous. We can see this because if we take the same expression a+b·a, it has two left most derivations.

   1. `Z → Z+Z → a+Z → a+Z·Z → a+b·Z → a+b·a`
   2. `Z → Z·Z → Z+Z·Z → a+Z·Z → a+b·Z → a+b·a`

   We can now define a similar language, but one that can be unambiguous. One way we can do this is by forcing asymmetry in the CFG, by introducing stricter variables. We take care of the precedence of addition over multiplication by introducing some extra variable which would handle multiplication.

   New Production rules as -

   ```
   Z → I | (Z) | Z+T | Z·Z;
   T → I | T * F
   F → I
   I → ε | a | b
   ```

This is a grammar which can handle expressions like **"a", " a* ", "a · b", "a · (a + b)", unambiguously.**