DLX Debugger

The DLX Debugger is designed for both debugging the target program and its compiler. A debug session requires 3 files:

1. The compiled program binary
2. The source code file
3. A third file containing debug symbols and comments

Usage

Run the debugger from the jar file with the following command:

```
java -jar debugger.jar [-e] [-s <source-file>] [-d <debug-file>] <program-file>
```

Where the options are:

-e Echo each assembly instruction to the shell as it is executed.

-s Optionally specify the source filename; if omitted, `<program-file>.dlx` is used.

-d Optionally specify the debug symbol filename; if omitted, `<program-file>.dbg` is used.

Examples:

```
java -jar debugger.jar -e src/test001
java -jar debugger.jar -s src/test001.txt -d dbg/test001.dbg bin/test001
```

Debug Symbols

A debug session requires a file containing debug symbols for the program. The file content is plain text, and must contain three sections, each beginning with the name of the section on a line by itself.

Section **.Lines**: on a single line, list the pairing of assembly instructions to source code lines in pairings:

```
(<source-line>,<assembly-instruction-index>)
```

For example:

```
.Lines
(25,5) (28,12) (29,14) (32,88)
```

This section informs the debugger how to highlight the code as it steps through the program instructions. When it reaches assembly instruction 5, it will highlight a block of assembly instructions 5-11, because the *next* block begins at 12. Similarly, it will highlight source code lines 25-27, because the *next* block of source lines begins at 28.

Section **.Heap**: List the heap allocations by variable name and allocation offset:

```
<offset>:<name>
<offset>:<name>
...
```

For example:

```
.Heap
0:digits
4:length
8:list[10]
48:i
...
```

Section **.Comment**: List the operational comment for each assembly instruction:

```
<index>:<comment>
<index>:<comment>
...
```

For example:

```
.Comment
0:Allocate 928 bytes on the heap
1:Set the stack pointer
2:Initialize variable length
3:Initialize variable digits
4:Save caller's return address
5:Save caller's frame pointer
6:Set the callee frame pointer
7:Call initialize()
8:Pop return value
9:Clear the stack
10:Restore caller's frame pointer
11:Restore caller's return address
12:Call print()
...
```

Memory Layout

The debugger assumes the head is at the end of the memory space, and the stack begins at the tail of the heap and of course goes in the other direction. To use a different memory layout, you will need to hack the debugger's source code, which is included in the jar.

Features

The debugger begins with a session ready to execute from the second block of assembly instructions. It automatically steps over the first block of `main`, which sets up the heap and stack pointers (allowing the data panes to initialize).

- Control execution of the target program using the buttons at the bottom of the debugger window.
- Change the execution granularity to assembly instructions by checking the "Assembly" checkbox.
- Set a breakpoint by clicking a line in the assembly window. Click again to clear it.
- Clear all breakpoints with the "Clear" button.
- Disable all breakpoints with the "Disable" checkbox.
- Correlate a line of source code with a line of assembly by clicking the source line.
- Panes and columns can be resized by dragging the dividers.